# Getting Started with Synergy/DE

## Version 9.1.3

Printed: March 2008

The information contained in this document is subject to change without notice and should not be construed as a commitment by Synergex. Synergex assumes no responsibility for any errors that may appear in this document.

The software described in this document is the proprietary property of Synergex and is protected by copyright and trade secret. It is furnished only under license. This manual and the described software may be used only in accordance with the terms and conditions of said license. Use of the described software without proper licensing is illegal and subject to prosecution.

© Copyright 2001–2008 by Synergex

Synergex, Synergy, Synergy/DE, and all Synergy/DE product names are trademarks of Synergex.

ActiveX, JScript, and Windows are registered trademarks of Microsoft Corporation.

Serena and ChangeMan are registered trademarks and Builder is a trademark of Serena.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems Inc., in the United States and other countries.

SlickEdit and Context Tagging are registered trademarks of SlickEdit, Inc.

Workbench's context-tagging support was developed using ANTLR from MageLang Institute.

All other product and company names mentioned in this document are trademarks of their respective holders.

DCN GS-01-9105

Synergex
2330 Gold Meadow Way
Gold River, CA 95670 USA

**http://www.synergex.com**
phone 916.635.7300
fax 916.635.6549

# Contents


## Preface

About this manual ix
Manual conventions ix
Other useful publications x
Product support information x
Synergex Professional Services Group xi
Comments and suggestions xi

## 1 What Is Synergy/DE?

What Is Synergy/DE? 1-2
An underlying philosophy 1-2
Advantages of Synergy/DE 1-2
Elements of Synergy/DE 1-4
Professional Series 1-4
Connectivity Series 1-6
*xf*Series 1-6
Developing an Application 1-7
Developing a distributed application 1-8
Using the Documentation 1-9
Getting Started with Synergy/DE 1-9
Online Help 1-9
Synergy/DE reference manuals and user's guides 1-9
Other documents 1-10

## 2 Developing Your Application in Workbench

What Is Professional Series Workbench? 2-3

Where Do You Begin? 2-4

- Starting Workbench 2-4
- Defining the startup environment 2-5
- Running Workbench on a Terminal Services machine or in a shared configuration 2-7

Accessing Synergy/DE Tools 2-8

Setting Up Your Development Environment 2-10

- Creating a workspace 2-10
- Creating a project 2-11

Editing Synergy Code with the Workbench Editor 2-15

- Setting up automatic code formatting and completion 2-15
- Editing a file 2-16
- Working with tag files 2-19
- Moving between files in your project 2-21
- Displaying online Help 2-21

Generating Synergy Code Segments 2-22

- Using aliases 2-22
- Using code templates 2-26

Analyzing Your Code 2-28

- Viewing a call tree of external routines 2-28
- Viewing where a method is called 2-29
- Browsing an ActiveX control 2-30

Compiling, Building, Running, and Debugging 2-31

- Checking compilation errors 2-31
- Debugging a project 2-31

Customizing Your Development Environment 2-33

- Customizing the way a project is opened 2-33
- Customizing and adding commands 2-34
- Adding file extensions to Workbench 2-36
- Customizing keyword color coding 2-37
- Changing the tagging delay 2-37
- Turning tagging off 2-38
- Interfacing with version control tools 2-38

Understanding the .NET Framework version used by Workbench 2-38
Copying customization settings 2-39
Using Workbench for Non-Windows Development 2-41
Using NFS-based mapped drives 2-41
Using FTP 2-42

## 3 Setting Up Your Repository

What Is Repository? 3-2
Starting Repository 3-2
Getting help 3-2
Displaying a list of valid data for a field 3-2
Defining a Record Layout for Use in an Application 3-3
Defining a structure 3-3
Defining fields 3-4
Saving your structure 3-6
Defining User Interface Characteristics 3-7
Defining a structure 3-7
Defining how input is redisplayed in a field 3-8
Defining fields 3-9
Defining field attributes 3-9
Defining Files for ReportWriter 3-15
Defining a structure 3-15
Defining fields 3-15
Determining how each field will be displayed in a report 3-15
Defining a format 3-16
Defining a key to your record 3-16
Defining a relation between two structures 3-18
Defining a file 3-19
Assigning a structure to a file 3-20
Defining a Database Schema for *xf*ODBC 3-21

## 4 Designing Your User Interface

Important Terminology 4-2
What Is Composer? 4-3
Starting Composer 4-3
What's on your Composer screen? 4-3
Using Help 4-6
Using Composer 4-8
Designing an input window using repository fields 4-8
Designing an input window from scratch 4-12
Saving your work 4-16
Compiling your script 4-17
Exiting Composer 4-18

## 5 Programming in Synergy Language

What Is Synergy Language? 5-2
Creating a source file 5-2
Structure of a Synergy Language program 5-3
Compiling, Linking, and Running Your Program 5-5
Compiling your program 5-5
Creating object libraries 5-6
Linking your program 5-6
Running your program 5-8
Debugging Your Program 5-9
Saving and restoring debugger settings 5-9
Debugging with bounds checking 5-10
Advanced Features 5-12
Using dynamic memory 5-12
Dispatching routines dynamically 5-17
Programming Tips 5-18
Referencing data indirectly 5-18
Comparing data 5-20
Manipulating dates 5-21
Using compile-time definitions 5-23
Using integer data 5-23
Using CASE vs. USING 5-23

## 6 Implementing Your User Interface with UI Toolkit

What Is UI Toolkit? 6-3
Performing terminal I/O 6-3
Starting UI Toolkit 6-4
The Toolkit screen 6-4
Letting Toolkit manage your files 6-5
Using event-style programming 6-6
Including tools.def 6-6
Using variables for identifiers 6-7
Managing Display Levels with Environment Processing 6-8
Local and global screen components 6-9
Managing Window Libraries to Store and Retrieve Display Components 6-10
Benefits of window libraries 6-10
Specifying a window library 6-10
Creating Script Files and Window Libraries 6-11
Creating and using menu columns, windows, and lists 6-11
Implementing Online Help 6-35
Implementing native Toolkit help 6-36
Organizing Your Display with Tabbed Dialogs 6-39
Using Composite Windows to Combine Windows and Lists 6-42
Using Methods 6-45
Using input methods 6-45

## 7 Accessing Data Remotely with *xf*Server

What Is a Client/Server System? 7-2
A basic client/server model 7-2
A multi-tier client/server model 7-3
Benefits of a client/server system 7-4
What Is *xf*Server? 7-5
How the *xf*Server system operates 7-5

## 8 Accessing Logic Remotely with *xf*ServerPlus

Overview 8-2

What Are *xf*ServerPlus and *xf*NetLink? 8-3

*xf*ServerPlus 8-3

*xf*NetLink Synergy Edition 8-3

*xf*NetLink Java Edition 8-4

*xf*NetLink COM Edition 8-6

*xf*NetLink .NET Edition 8-6

Design Considerations 8-8

Separating the user interface from application logic 8-9

Separating data access from application logic 8-9

Using ELBs or shared images 8-10

Handling errors 8-10

Guidelines to improving performance and resilience 8-11

## Glossary

## Index

# Preface

## About this manual

*Getting Started with Synergy/DE* provides a technical overview of all components of Synergy/DE™, as well as a task-oriented "cookbook" for using Synergy/DE. It walks you through the steps required to develop a basic Synergy™ application—and tells you what else you can do and where to get more information if you want to go beyond the basics.

Because we assume that you are familiar with programming in some language, this guide does not cover basic principles of programming. Instead, it includes language features specific or integral to developing in Synergy/DE.

## Manual conventions

Throughout this manual, we use the following conventions:

- In code syntax, text that you type is in `Courier` typeface. Variables that either represent or should be replaced with specific data are in *italic* type.
- Optional arguments are enclosed in *[*italic square brackets*]*. If an argument is omitted and the comma is outside the brackets, a comma must be used as a placeholder, unless the omitted argument is the last argument in a subroutine. If the comma is inside the brackets and an argument is omitted, the comma may also be omitted.
- Arguments that can be repeated one or more times are followed by an ellipsis…
- A vertical bar (|) in syntax means to choose between the arguments on each side of the bar.
- Data types are **boldface**. The data type in parentheses at the end of an argument description (for example, (**n**)) documents how the argument will be treated within the routine. An **a** represents alpha, a **d** represents decimal or implied-decimal, an **i** represents integer, and an **n** represents numeric (which means the type can be **d** or **i**).

**WIN**

Items or discussions that pertain only to a specific operating system or environment are called out with the name of the operating system.

- To “enter” data means to type it (or highlight it, in the case of a selection window entry) and then press ENTER. (“ENTER” refers to either the ENTER key or the RETURN key, depending on your keyboard.)
- When you are instructed to “select an entry from the menu,” either press the shortcut for that entry, or pull down the appropriate menu, highlight the entry, and press ENTER.
- To “Exit the window” on Windows, click the OK button or press the Exit shortcut.

  On UNIX and OpenVMS, press the Exit shortcut, or press ENTER from the last field, and then ENTER again from the **Make changes or press RETURN to complete input** message.

## Other useful publications

- *Repository User’s Guide*
- *Synergy Language Reference Manual*
- *Synergy Language Tools*
- *Environment Variables and System Options*
- *UI Toolkit Reference Manual*
- *Installation Configuration Guide*
- *Developing Distributed Synergy Applications: Using xfNetLink and xfServerPlus*
- Professional Series Workbench online Help system
- S/DE Composer online Help system

## Product support information

If you cannot find the information you need in this manual or in the publications listed above, you can call the Synergy/DE HelpDesk Support department at (800) 366-3472 (in North America) or (916) 635-7300. To purchase a HelpDesk Support agreement, contact your Synergy/DE account manager at the above phone numbers.

Before you contact us, make sure you have the following information:

- The version of the Synergy/DE product(s) you are running
- The name and version of the operating system you are running
- The hardware platform you are using
- The error mnemonic and any associated error text (if you need help with a Synergy/DE error)
- The statement at which the error occurred
- The exact steps that preceded the problem
- What changed (for example, code, data, hardware) before this problem occurred

- Whether the problem happens every time and whether it is reproducible in a small test program
- Whether your program terminates with a traceback, or whether you are trapping and interpreting the error

## Synergex Professional Services Group

If you would like assistance implementing new technology or would like to bring in additional experienced resources to complete a project or customize a solution, Synergex™ Professional Services Group (PSG) can help. PSG provides comprehensive technical training and consulting services to help you take advantage of Synergex's current and emerging technologies. For information and pricing, contact your Synergy/DE account manager at (800) 366-3472 (in North America) or (916) 635-7300.

## Comments and suggestions

We welcome your comments and suggestions for improving this manual. Send your comments, suggestions, and queries, as well as any errors or omissions you've discovered, to **doc@synergex.com**.

# 1

# What Is Synergy/DE?

Welcome to Synergy/DE version 9, the next generation of Synergy tools designed with programmer productivity and flexibility in mind.


### What Is Synergy/DE? 1-2

Defines the underlying philosophy and advantages of developing in Synergy/DE.

### Elements of Synergy/DE 1-4

Briefly describes each component of Synergy/DE.

### Developing an Application 1-7

Illustrates the products and steps you can use to create a Synergy application.

### Using the Documentation 1-9

Describes the reference manuals, user's guides, and online Help that are included with the Synergy/DE products.

# What Is Synergy/DE?

Synergy/DE is a full suite of development tools that provides everything you need to create powerful, event-driven business applications. It includes tools to write and process your applications, design and support your user interface, manage your data from one central location, and create comprehensive reports.

## An underlying philosophy

If you were building a house, you would need many tools, ranging from the very powerful to the very basic. Having these tools readily available would reduce the time and maximize the efficiency with which you could complete the job. This same principle applies to developing a business application: Well-chosen, accessible tools make the job a lot easier. Using the right tool for the right job is the fundamental philosophy of Synergy/DE.

Using Synergy/DE, you can focus most of your resources on your vertical or custom application, because Synergy/DE takes care of your systems-level development.

## Advantages of Synergy/DE

Synergy/DE enables you to

- take advantage of one of the most highly scalable distributed technologies. Unique Synergy multi-tier technology enables you to scale your application from one stand-alone machine to an enterprise-wide system hosting thousands of users.
- open your Synergy business logic and data to a variety of thin client technologies. Using Synergy/DE *xf*NetLink and *xf*ServerPlus, you can build distributed solutions that enable Web browsers, Visual Basic front-end applications, Active Server Pages, or Synergy thin clients to access Synergy routines and database files.
- maintain database independence. Third-party applications can access your Synergy data, and your Synergy application can access popular, third-party SQL databases.
- easily create and maintain your application. “Draw” and maintain user interface elements in a visual environment. Easily add new features to accommodate your customers’ changing needs. Quickly customize and localize your user interface.
- keep your applications portable. Develop your applications in any environment—Windows, virtually all UNIX platforms, OpenVMS—and in most cases deploy in other environments by simply relinking.
- save time. Save development and maintenance time by creating and reusing object and executable libraries. Call routines or share information with applications written in other languages.

- use functional depth. Develop complete applications by working only at the high level, such as in Composer or by calling environment-supplied subroutines, or drop down to the core level where you have complete flexibility to customize all aspects of your application.
- manage the development process. Eliminate duplication of work by reusing supplied and custom libraries and centralizing all data definitions where the entire development team can access them and ensure consistency between efforts.
- reduce maintenance. Centralizing data definitions and libraries means that changes can be made in one central location rather than throughout thousands of lines of codes.

# Elements of Synergy/DE

The following Synergy products comprise Synergy/DE:

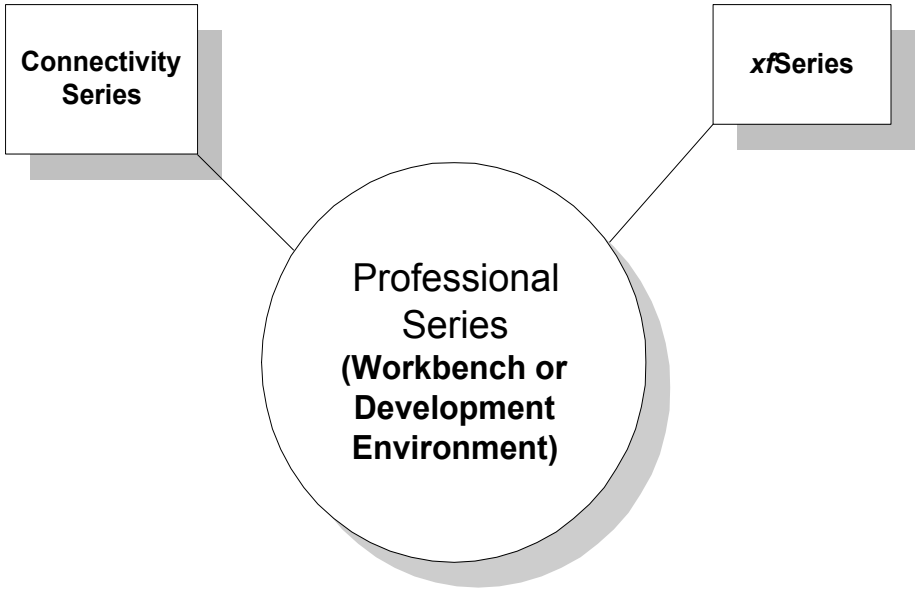



*Figure 1-1. The components of Synergy/DE.*

## Professional Series

Both Professional Series Workbench and Professional Series Development Environment include the components listed below. In addition, Professional Series Workbench provides an interface that not only launches all of these components but features a Synergy Language–sensitive visual editor as well as project management tools. (Professional Series Workbench is only available on Windows.)

### UI Toolkit

UI Toolkit gives you all you need to create your application's user interface, including an extensive set of subroutines to handle your environment maintenance and menu, input, text, selection, and list processing.

## Composer (Windows only)

Composer is an interactive user-interface design tool. You “draw” windows and input windows just the way you like, graphically assigning properties that are immediately reflected on your screen, and Composer generates the appropriate script commands to define your objects. Composer also accepts predefined input windows and fields from Repository, so you can view and modify them graphically.

## Repository

Repository is the tool you use to define and maintain the data that will be used in your application or in reports that can be created with Synergy/DE ReportWriter or *xf*ODBC. It provides a method of managing data structures and fields in one centralized location, so that you can modify a field throughout your application simply by changing one definition in your repository.

## Synergy Language

Synergy Language is a high-level business programming language that encourages structured, system-independent, modular code. Beyond the basic statements that specify processing to be performed, Synergy Language includes a large number of prewritten external subroutines and functions, a source-level debugger, graphical user interface support, and a fast and efficient system-independent file structure.

## Synergy DBMS

The Synergy DBMS system includes everything needed to create and manage high-speed, keyed access and ordered sequential access databases.

## ReportWriter

ReportWriter is an end-user application that retrieves, integrates, and consolidates data into columnar, ad hoc reports. You can customize and include ReportWriter as part of your application, either predefining reports for your customers to run or enabling them to design their own reports.

## Synergy Runtime

Synergy Runtime supports the actions requested by your Synergy/DE applications during execution.

# Connectivity Series

## SQL Connection and Synergy database drivers

SQL Connection is Synergy/DE's SQL API that works in conjunction with the appropriate database driver to enable a Synergy application to access SQL data sources (typically RDBMSs). Synergy/DE provides database drivers that enable SQL Connection access to popular databases such as Oracle and SQL Server.

## SQL OpenNet

SQL OpenNet provides the middleware needed if your Synergy SQL application will be accessing remote RDBMSs (residing on a remote computer on your network).

## *xf*ODBC

*xf*ODBC opens Synergy DBMS data to any third-party, 32-bit ODBC application. It supports version 2.5 of the ODBC API (level 1) and uses SQL OpenNet for remote data access.

# *xf*Series

## *xf*Server

*xf*Server processes network client requests for Synergy DBMS data. Residing on the same computer as the Synergy databases, the server can serve data to any Synergy application running with a Synergy runtime (version 6.1 and higher).

## *xf*ServerPlus

*xf*ServerPlus handles the remote execution of Synergy routines.

## *xf*NetLink

*xf*NetLink provides four methods for accessing Synergy logic with an assortment of clients:

- *xf*NetLink Synergy Edition, which is a set of routines that work in conjunction with *xf*ServerPlus to execute Synergy routines stored on a remote machine
- *xf*NetLink Java Edition, which works in conjunction with Sun Microsystems' Java™ programming language
- *xf*NetLink COM Edition, which enables a client to call Synergy routines residing on a remote server machine using a standard COM calling mechanism
- *xf*NetLink .NET Edition, which works in conjunction with Microsoft's .NET Framework SDK

# Developing an Application

Figure 1-2 illustrates how you can use Professional Series to develop your application.

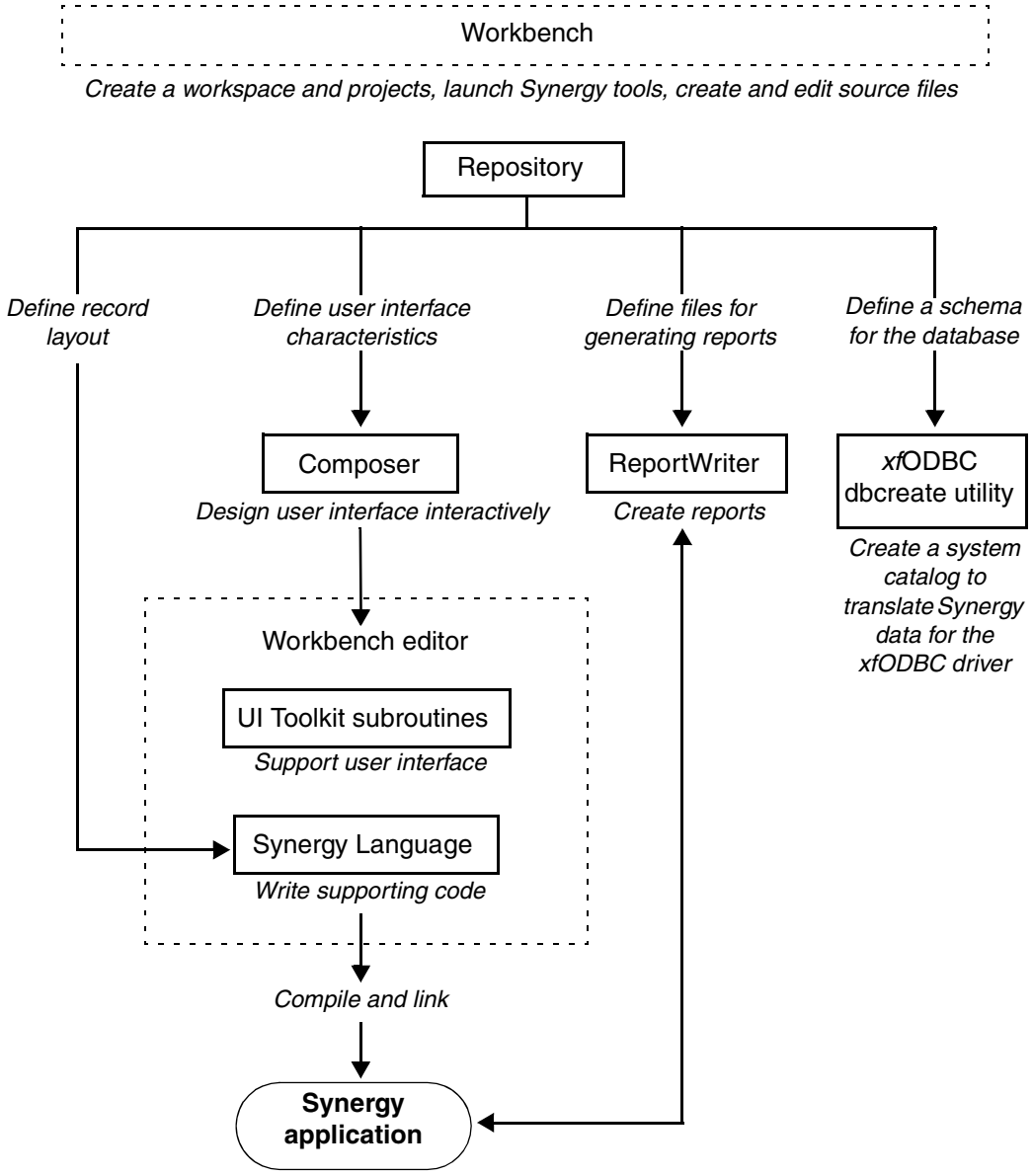



*Figure 1-2. Developing an application with Synergy/DE.*

## Developing a distributed application

Synergy/DE also provides the capability to design and deploy your Synergy application in two-tier or multi-tier systems. See chapter 8, "Accessing Logic Remotely with xfServerPlus." The *Developing Distributed Synergy Applications* manual provides additional information about building a distributed computing system.

# Using the Documentation

## *Getting Started with Synergy/DE*

This manual is primarily intended for experienced developers who are new users of Synergy/DE tools. It provides a technical overview of all components of Synergy/DE, as well as a task-oriented "cookbook" for developing with Professional Series. Like a tutorial, this guide walks you through the basic tasks (illustrated in figure 1-2) that you will perform as you develop your Synergy application. It also discusses client/server and web development.

## Online Help

Composer has a complete online Help system, including context-sensitive help. The Composer Help system provides online instructions for performing any task in Composer. You can search for information about any topic and then display that information, with hypertext links to additional, related information, on the screen. The context-sensitive help informs you to how to proceed from the current point or defines your current options. Workbench also has an online Help system, as do many of the Synergy/DE utilities.

## Synergy/DE reference manuals and user's guides

Here's a list of other manuals that will assist you in using Synergy products:

- *Synergy Language Reference Manual*

  Describes the syntax of all statements, functions, subroutines, compiler directives, utilities, and environment variables included in Synergy Language. Also documents other features of the Language, such as program organization, expressions, indexed sequential access method (ISAM), system options, compiler and runtime errors, the message controller, and the debugger.

- *Synergy Language Tools*

  Describes the programming tools that are used with Synergy Language: the compiler, linker, librarian, and runtime; the debugger; the Synergy DBMS file management system; and general utilities. Also documents the Synergy Language error messages.

- *Environment Variables and System Options*

  Lists and discusses all of the environment variables and system options available in Synergy/DE.

- *UI Toolkit Reference Manual*

  Describes the syntax of all Toolkit subroutines and utility programs, which are used to create and maintain state-of-the-art user interfaces.

- *Repository User's Guide*

  Explains how to define data in Repository and describes its utility functions. Also describes the syntax of Synergy Data Language statements and Repository information-retrieval subroutines.

- *ReportWriter User's Guide*

  Explains how to use ReportWriter to access and organize your data into useful reports. Includes a tutorial and describes the ReportWriter utility functions.

- *SQL Connection Reference Manual*

  Describes how to access SQL databases from Synergy Language applications.

- *Professional Series Portability Guide*

  Discusses the functions, requirements, and processes that are unique to each of the platforms on which Synergy/DE runs, so that you can design your application to be as portable as possible.

- *xfODBC User's Guide*

  Describes how to access Synergy databases from ODBC applications.

- *Installation Configuration Guide*

  Provides information about using License Manager to license your Synergy products, configuring a client/server system with *xf*Server, and installing and configuring Connectivity Series products. Also includes general information on Synergy/DE installation and operating system requirements.

- *Developing Distributed Synergy Applications: Using xfNetLink and xfServerPlus*

  Describes how to build a distributed Synergy application with a Synergy, Java, COM-compliant, or .NET front-end that enables you to access Synergy routines and data remotely.

## Other documents

The following migration guides and white papers are available on the Online Manuals CD:

- *Synergy/DE Quick Migration Guide*
- *Migrating Your Application to Windows*
- *Modularizing Your Synergy Code: The First Step to Distributed Computing*

# 2

# Developing Your Application in Workbench

This chapter introduces you to the Professional Series Workbench, a visual development environment that includes the Professional Series Development Environment toolset, an integrated launchpad that brings these tools together, Synergy Language–sensitive code editing capabilities, and project management tools. Professional Series Workbench is only available on Windows.


### What Is Professional Series Workbench? 2-3

Defines Professional Series Workbench.

### Where Do You Begin? 2-4

Describes how to set up the startup environment and how to run Workbench.

### Accessing Synergy/DE Tools 2-8

Explains how to launch Synergy/DE tools from Workbench.

### Setting Up Your Development Environment 2-10

Describes the benefits of using workspaces and projects and explains how to create them.

### Editing Synergy Code with the Workbench Editor 2-15

Explains how to edit a file, automate code formatting and completion based on your own company style, display popup help for routine syntax, and work with tag files.

### Generating Synergy Code Segments 2-22

Describes how to use aliases and code templates to quickly generate Synergy Language/UI Toolkit code for some of the more common code structures, including your own method routines.

### Analyzing Your Code 2-28

Describes the analysis tools that can help you identify which routines are called by a specified routine and which routines call a specified routine.

### Compiling, Building, Running, and Debugging 2-31

Describes how to compile, build, and run your project from Workbench, as well as how to customize compile, build, debug, and execute commands for individual projects.

### Customizing Your Development Environment 2-33

Discusses how to customize the way a project is opened, define script extensions other than **.wsc**, add your own commands to the Workbench menu, add or remove buttons from the Workbench toolbar, change the tagging delay or turn tagging off completely, access the version control tool of your choice, and copy or save your customization settings.

### Using Workbench for Non-Windows Development 2-41

Describes how you can use NFS-based mapped drives or FTP to make Workbench your primary editor on non-Windows systems.

# What Is Professional Series Workbench?

Professional Series Workbench is an enhanced version of the Professional Series Development Environment (PSDE). Workbench makes moving between Repository, Composer, and other Synergy/DE components easier as you develop your applications. Workbench's smart, fully customizable language-sensitive editor provides color coding, automatic indenting, standardized code templates, project tag capability for recognizing and accessing routine code, and many other code-automating features. It enables you to take advantage of automated script compiling and many other features that can streamline your development efforts.

In short, Workbench can help you spend less time writing and debugging, ensure standard coding practices, and speed your applications to market.

The Workbench editor is based on SlickEdit® technology from SlickEdit, Inc.

# Where Do You Begin?

## Starting Workbench

- From your SynergyDE folder on the Start menu, select Workbench.

The Workbench desktop looks like this:

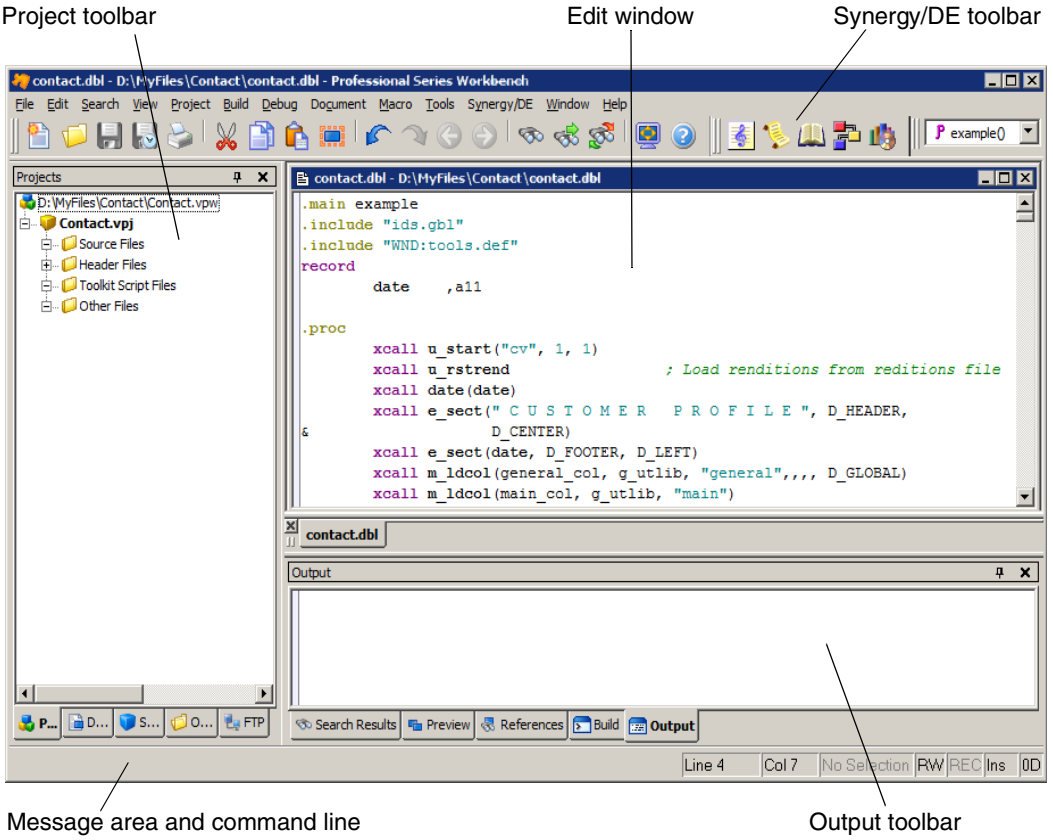



*Figure 2-1. The Workbench desktop.*

Running Workbench for the first time after installation runs the **update_synergy** command, which initializes Workbench and creates the SlickEdit configuration directory, C:\Documents and Settings*user*\My Documents\My SlickEdit Config*version*, where *user* is the user name of the user who is currently logged in and *version* is the current version of SlickEdit in the format *x.x.x* (for example, 10.0.2).

# Defining the startup environment

Setting environment variables and initialization settings enables you to control program functionality externally without modifying your code. For a more complete discussion of the advantages of environment variables, refer to "Referencing data indirectly" on page 5-18. For a list of all environment variables and initialization settings available to Synergy/DE and instructions on how to set them, refer to the "Environment Variables" chapter of *Environment Variables and System Options*.

You can define environment variables and initialization settings either in the environment, in the **synergy.ini** file, or in the Open tab of Workbench's Project Properties dialog box. **Synergy.ini** is an initialization file that contains variables that affect the Synergy runtime and Synergy/DE development tools on Windows.

Variables are set in the following order:

- Globally, by selecting Start > Settings > Control Panel > System > Advanced > Environment Variables
- At a DOS prompt
- In the **synergy.ini** file (which is read when the project is opened)
- In the Open tab of the Project Properties dialog box (see "Customizing the way a project is opened" on page 2-33) or using **syn_set** or **syn_set_global** on the Workbench command line (see "Setting environment variables in Workbench" on page 2-6)

Environment variables set in the environment are available to Workbench when it is started. All appropriate initialization settings from the [synergy] section of **synergy.ini** are set in the order in which they are specified in **synergy.ini**. Settings in **synergy.ini** override any duplicate variables set at the environment level. We set the variables from the active **synergy.ini** when Workbench is started and again whenever a Synergy/DE project is opened. We restore the settings from the **synergy.ini** that was active when Workbench was started whenever a project is closed. Settings in a project's Open tab are specific to that project.

## Accessing the Workbench command line

Besides enabling you to set environment variables, the Workbench command line can be used to execute Workbench or operating system commands and to run other programs. You can display and move the cursor to the Workbench command line in one of two ways:

- With your cursor in the edit window, press ESC.
- Click in the message area at the bottom of the Workbench window. (See figure 2-2.)

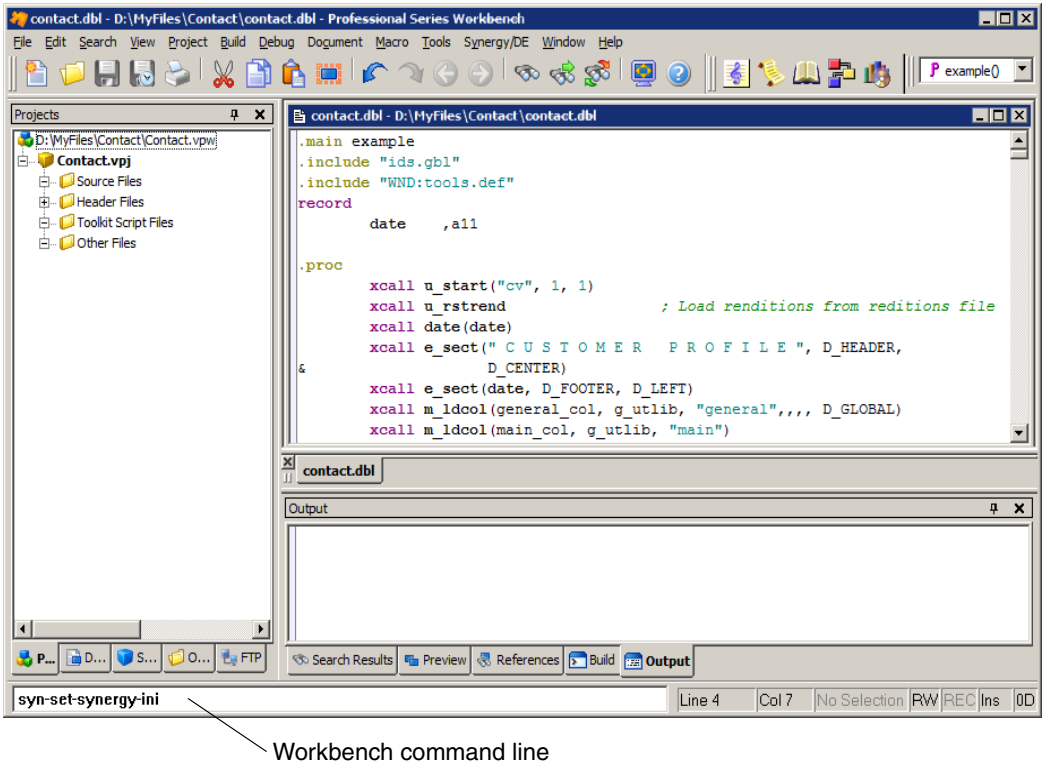



*Figure 2-2. Typing at the Workbench command line.*

## Setting environment variables in Workbench

To set an environment variable when a Synergy project is open,

- At the Workbench command line or in the Open tab of the Project Properties dialog box, enter the following command:

  `syn_set` *var=value*

  where *var* is the environment variable you want to set and *value* is the value you want to set it to. This command overrides any variable set globally, at a DOS prompt, or in **synergy.ini** while the current project is active. If *value* is empty, Workbench unsets *var* from the environment while the project is open. When the project is no longer active, the variable is unset (or reset to its original value if it was already set when Workbench was invoked).

To make Workbench load the environment variables from the [synergy] section of **synergy.ini** and then **synuser.ini**,

- At the Workbench command line or in the Open tab of the Project Properties dialog box, enter the following command:

  `syn_set_synergy_ini` *[path]*

  where *path* is the directory path to which the SFWINIPATH environment variable will be set. The environment variables will be loaded from the [synergy] section of the **synergy.ini** and **synuser.ini** files in that path. If *path* is not specified, Workbench uses the default path for **synergy.ini** and **synuser.ini** as defined in "Setting Environment Variables and Initialization Settings" in the "Environment Variables" chapter of *Environment Variables and System Options*. If **syn_set_synergy_ini** is not called in the project's Open tab, it will be called after all project open commands.

Environment variables set with **syn_set** after **syn_set_synergy_ini** is called are available in the environment for any command that Workbench spawns. Be aware, however, that the Synergy tools (**dbl**, **dblink**, etc.) may read the [synergy] section of **synergy.ini**, which will override settings made in the spawned environment when these tools are launched from Workbench. We recommend that you use **syn_set** to add environment variables based on values in **synergy.ini**, instead of trying to override variables that are set in **synergy.ini**.

We do not recommend using the SlickEdit **set** command to change the value of an environment variable.

To reset your environment variables back to what they were when you opened the project,

- At the Workbench command line, enter the following command:

  `syn_init_proj` *filename*

  where *filename* is the name of a file that will list the environment variables and initialization settings after they are reset. See "Accessing the Workbench command line" on page 2-5 if you need help displaying the Workbench command line.

## Running Workbench on a Terminal Services machine or in a shared configuration

On a Terminal Services machine or in a shared configuration, the **update_synergy** command only creates the SlickEdit configuration directory for the user who is currently logged in. For clients of a shared installation, it creates this directory on the client machine. For Terminal Services users, it creates this directory on the Terminal Services machine.

# Accessing Synergy/DE Tools

You can start the following tools from the Synergy/DE menu and/or toolbar:

- Composer
- Script
- Repository
- Method Definition utility
- Synergy/DE online manuals

To launch one of these applications (or to bring it to the forefront if it's already running), do one of the following:

- From the Synergy/DE menu, select the desired application.
- Click the desired toolbar button, as shown in figure 2-3.

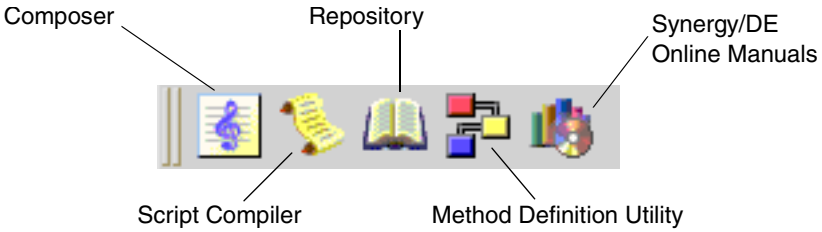



*Figure 2-3. The Synergy/DE toolbar.*

When Composer is selected, if a Synergy/DE project is open and a script file is active, Composer is launched with that script file. If no script file is active, or no script files exist in the project, the Launch Composer dialog is displayed, where you can select an existing script or create a new script to edit in Composer.

---

We do not recommend or support the maintenance of multiple versions of Workbench (or Composer). If you are planning to maintain multiple versions of Synergy/DE, you must remove the older versions of Workbench (and Composer) before installing the newest version.

---

The following utilities are also available from the Utilities submenu:

- ActiveX Diagnostic utility
- Script-to-Repository Conversion utility
- Synergy Control Panel

- Method Definition utility
- Synergy Type Library Configuration utility
- Synergy .NET Configuration utility
- SMC/ELB Comparison utility
- Variable Usage utility

Depending on what type of project is open, additional utilities may be available from the Build menu.

To run the Synergy Prototype utility (**dblproto**), which generates prototypes for the source code in a Synergy/DE project, select Build > Generate Synergy Prototypes.

# Setting Up Your Development Environment

## Creating a workspace

We suggest that you group related projects together by creating a workspace. Tagging at the workspace level enables you to have context menu help for routines and classes across projects. The data for each workspace is stored in a text file that has the extension **.vpw**.

1. From either the File or Project menu, select New and then click the Workspaces tab. The Workspaces tab of the New dialog box is displayed. (See figure 2-4.)

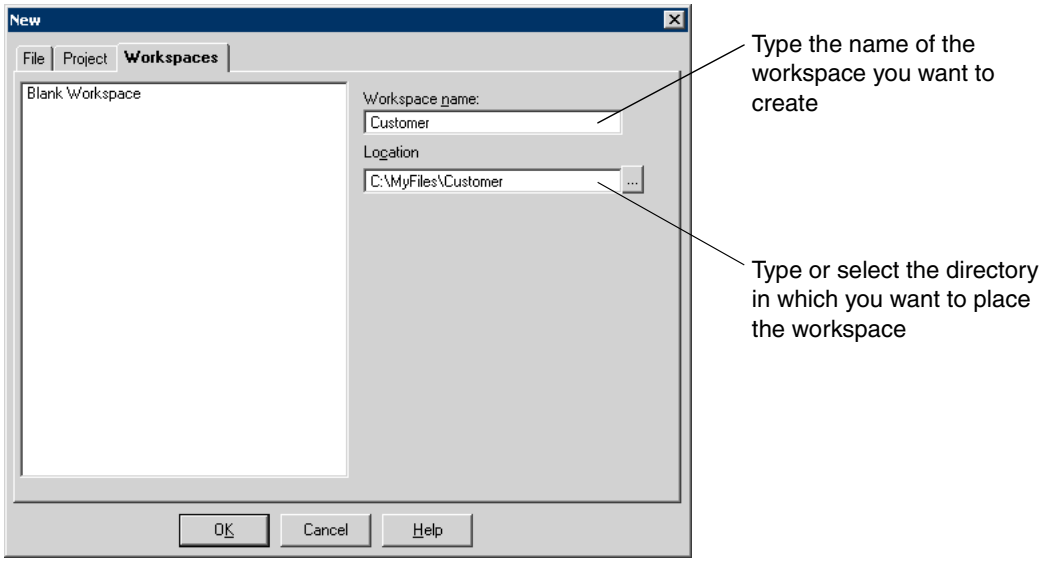



*Figure 2-4. Creating a workspace.*

2. Enter the desired information in each input field and then exit the dialog box.

   You can also create a workspace when you create a project. See "Creating a project" on page 2-11.

### Opening or closing a workspace

To open a workspace,

1. From the Project menu, select Open Workspace. The Open Workspace dialog box is displayed.
2. Type or select the name of the workspace you want to open.

   To close the current workspace, select Close Workspace from the Project menu.

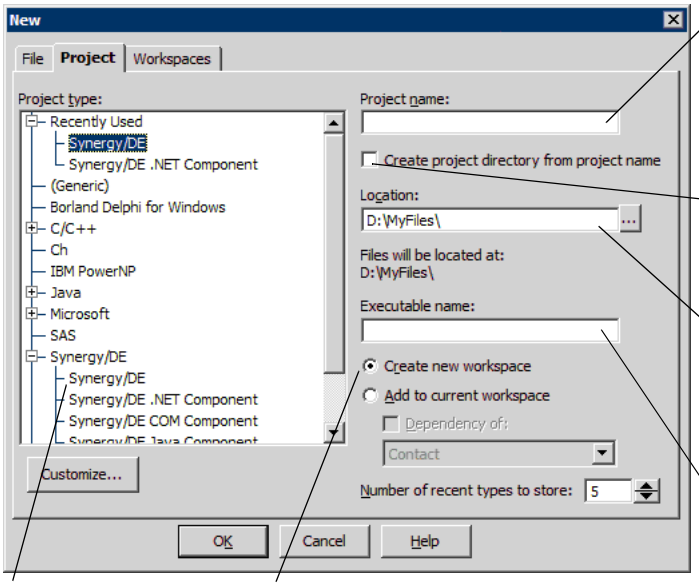
# Creating a project

Projects enable you to specify collections of files and build parameters that operate on those files. They facilitate tagging files, restoring edit sessions, compiling from Workbench, customizing builds, loading files, and managing version control, and they enable you to define environment variables specific to a project. We recommend that you create a project if you want to use one or more of these features.

You can create Synergy/DE, Synergy/DE COM Component, Synergy/DE Java Component, or Synergy/DE .NET Component projects (among others) in Workbench. For instructions on creating a COM, Java, or .NET component project, see the appropriate chapter of the *Developing Distributed Synergy Applications* manual: "Creating a Synergy/DE COM Component Project" in the "Creating Synergy COM Components" chapter, "Creating a Synergy/DE Java Component Project" in the "Creating Java Class Wrappers" chapter, or "Creating a Synergy/DE .NET Component Project" in the "Creating Synergy .NET Assemblies" chapter.

To create a regular Synergy/DE project,

1. From the Project menu, select New. The Project tab of the New dialog box is displayed. (See figure 2-5.)




Type the name of your project, using alphanumeric characters. Workbench assigns a filename extension of **.vpj** to each Synergy/DE project

Select Create project directory from project name to create a directory with the same name as the project

Type or select the directory in which you want your project to be saved. If the directory you specify does not exist, you will be prompted to create it

Type the executable name for this project or accept the default, which is the name of the project with a **.exe** extension. This field can be left blank if the project is building more than one executable

Select Synergy/DE

Select Create new workspace to create a new workspace for this project or Add to current workspace to add this project to the current workspace

*Figure 2-5. Creating a Synergy project.*

2. Enter the desired information in each input field and then click OK.

   The Project Properties For *filename* dialog box is displayed. (See figure 2-6.) From here you can add files to a project and define other project settings.

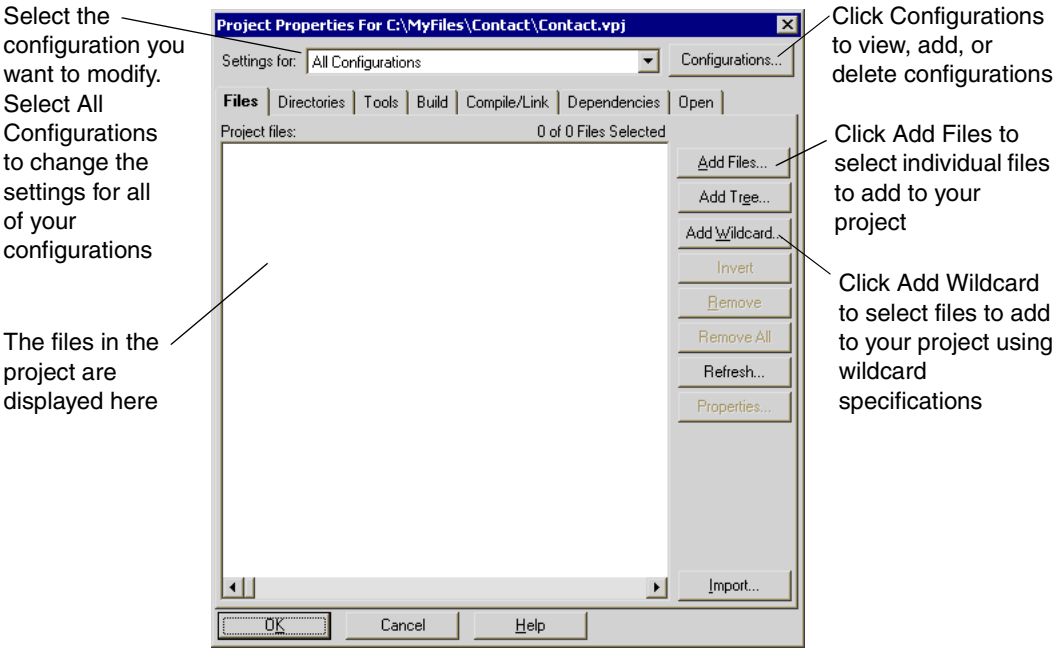



*Figure 2-6. Adding files to a project.*

3. From the Files tab of the Project Properties For *filename* window, click Add Files to add files to your project. The Add Source Files dialog box is displayed. (See figure 2-7.) (You can also add files by clicking Add Wildcard.)
4. Select the files you want to add and click Open. The Project Properties For *filename* dialog box is redisplayed with the files you selected listed in the Project files field.
5. If you want to add a new configuration to this project, see "Defining multiple configurations of a project" on page 2-13 below.
6. Exit the dialog box when you're done, unless you want to define other project properties.

   To add additional files (including ActiveX controls) later, right-click on the project in the Projects tab of the project toolbar and select Add Files. The dialog box in figure 2-6 is redisplayed.

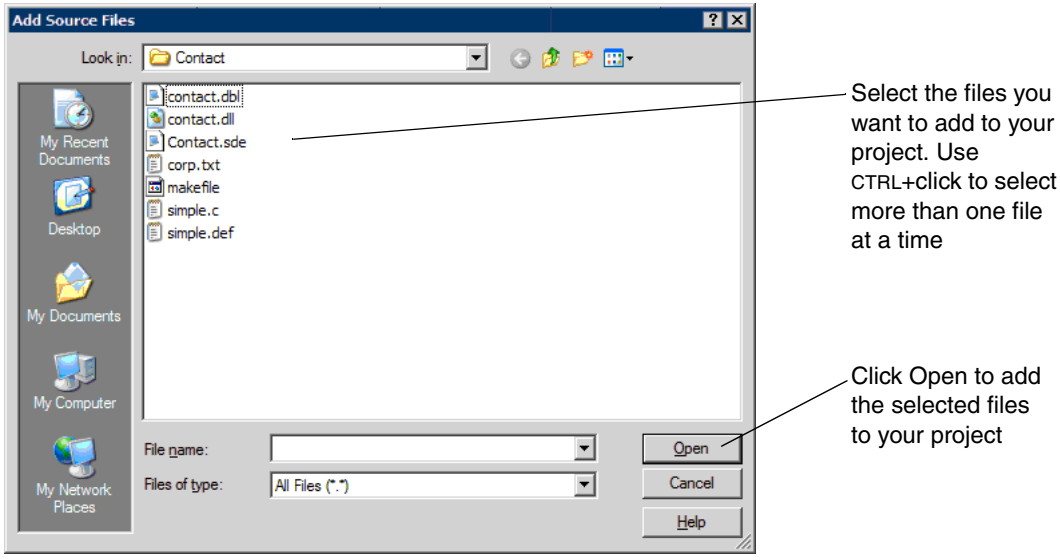



*Figure 2-7. Adding files to a project.*

## Defining multiple configurations of a project

The most common use of project configurations is to enable you to have a debug version and a release version of your project without defining two different projects. To accomplish this, you can either configure your build commands to place the object code in different directories or perform a rebuild after switching configurations. In addition, the build commands for the debug version need to specify any debug options. To add a configuration,

1. If you're in the Project Properties For *filename* dialog box, click the Configurations button. If not, select Configurations from the Build menu. The Project Configuration Settings dialog box is displayed.
2. Click New. The Create new configuration dialog box is displayed.
3. Enter the name of the configuration you want to add, and select a configuration whose settings you want to copy. Click OK when you're finished.
4. Highlight the configuration you want to make active and click the Set Active button. Click OK when you're finished.

   You can later change the active configuration from the Build menu by selecting Set Active Configuration and then selecting the desired configuration from the submenu.

## Adding a project to a workspace

You can add a project to a workspace either when you create that project or later. When you are creating a new project, select the Add to current workspace radio button on the Project tab of the New dialog box. (See figure 2-5.)

To add a project to a workspace after the project has been created,

1. From the Project menu, select Insert Project into Workspace. The Add Project to Workspace dialog box is displayed.
2. Type or select the name of the project file you want to add to the current workspace.

Sharing workspace and project files can cause project corruption or abnormal side effects. We strongly recommend that workspace and project files reside on your local hard drive.

# Editing Synergy Code with the Workbench Editor

Professional Series Workbench includes a language-sensitive visual editor that provides

- automatic code indentation.
- Synergy Language recognition.
- Synergy Language compiler integration.
- Context Tagging.
- repository awareness.
- context-sensitive routine help.

## Setting up automatic code formatting and completion

To set up or customize automatic code indentation and completion,

1. From the Tools menu, select Options > File Extension Setup, and click the Indent tab if it is not already selected.
2. In the Extension field, select **dbl**.
3. In the Indent style field, select Syntax indent so that the editor will indent your code according to Synergy Language syntax when Enter is pressed, and type the number of characters that each level should be indented.

   If Indent with tabs is checked, pressing Tab or Enter will indent with tabs rather than spaces. The value in the Tabs field sets the tabbing increment or individual tab stops.

We recommend that the Tabs value be either the same as or a multiple of the value of Syntax indent. If these values do not match, tabs will only be inserted when the cursor is indented past a tab location. We do not recommend specifying individual tab stops in the Tabs field.

The Workbench editor offers four styles of automatic code indentation to choose from for conditional statements (IF, CASE, USING), looping statements (WHILE, REPEAT, FOR), and BEGIN-END blocks. When you type such statements, they are automatically indented based on the style you have selected. For example, as you type BEGIN beneath an IF statement, both the BEGIN and the END statement (which you can specify should be added automatically when BEGIN is typed) are formatted and indented according to your chosen style. To set this up,

4. Click the DBL Options button. The DBL Options dialog box is displayed. (See figure 2-8.)
5. Enter the desired information in each input field and then click OK.
6. Click OK again to exit the Extension Options dialog box.

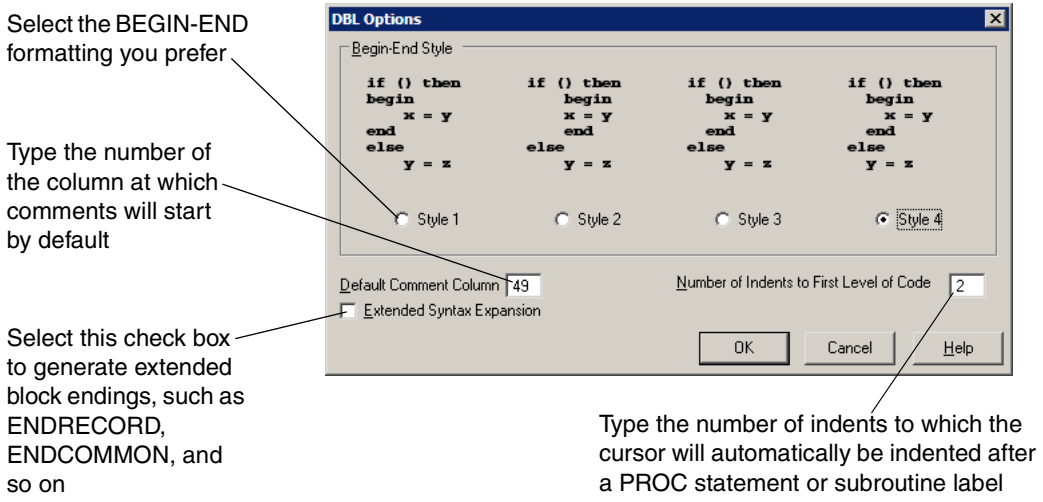

Select the BEGIN-END formatting you prefer
Type the number of the column at which comments will start by default
Select this check box to generate extended block endings, such as ENDRECORD, ENDCOMMON, and so on
DBL Options
Begin-End Style
Style 1
Style 2
Style 3
Style 4
Default Comment Column
49
Extended Syntax Expansion
Number of Indents to First Level of Code
2
OK
Cancel
Help
Type the number of indents to which the cursor will automatically be indented after a PROC statement or subroutine label

*Figure 2-8. Setting code formatting options.*

All of the above settings are global, rather than project-specific, settings.

## Editing a file

You can either create a new file or edit an existing one.

1. Do one of the following:

| To | Do this |
|---|---|
| Create a new file | Select File > New.<br>In the New dialog box, select **DBL** from the list of file types. You can also specify a filename, location, and whether to place the file in the current project. |
| Edit an existing file | Select File > Open.<br>In the Open dialog box, type or select the name of the file you want to edit. |

The file opens in the edit window. (See figure 2-1.)

2. If the file is untitled or if it does not have a **.dbl** extension, save your file as a **.dbl** file using the Save or Save As command.
3. Type or edit your code, taking advantage of keyword completion and syntax expansion (if desired) as explained in the sections below.
4. When you're done making edits, save your file again.

## Taking advantage of keyword completion

After you type a Synergy Language keyword, the color of the text changes to signify that the editor recognizes a keyword. If the Enable auto-completion and Keywords check boxes on the Auto-Complete tab of the Extension Options dialog are selected, when you type a partial keyword, a drop-down list of possible keywords is displayed. Use the up and down arrow keys to highlight the keyword or alias that you want and select it by pressing Enter or the spacebar. For instance, if you type "exit", the keywords EXITLOOP and EXITTRY are displayed in a drop-down list.

Compiler directives, although recognized as keywords, are not automatically completed in this way.

If you want to customize the color of your keywords and compiler directives, see "Customizing keyword color coding" on page 2-37.

To turn on keyword completion and set the minimum number of characters at which the editor will complete the keyword for you,

1. Select Tools > Options > File Extension Setup. The Extension Options dialog box is displayed.
2. In the Extension field, select **dbl**, and click the Auto-Complete tab if it is not already displayed.
3. Select the Enable auto-completion check box.
4. Select the Keywords checkbox.
5. In the Minimum prefix length field, type the minimum number of characters to type before the drop-down keyword list is displayed, and then click OK.

## Taking advantage of syntax expansion

When syntax expansion is enabled, Workbench will insert the corresponding block ending keyword(s) whenever you type a block beginning keyword followed by space at the end of a line. Examples of block beginning keywords are CLASS (ENDCLASS), FOR (FROM...THRU), and SUBROUTINE (PROC).

To enable alias expansion and to make Workbench automatically insert block-ending keyword(s) for the corresponding block-beginning keywords,

1. Select Tools > Options > File Extension Setup. The Extension Options dialog box is displayed.
2. In the Extension field, select **dbl**, and click the Indent tab if it is not already displayed.
3. Select the Syntax expansion check box.
4. In the Minimum expandable keyword length field, type the minimum number of characters to type before alias expansion can occur.

5. If you want to generate extended block endings, click DBL Options and select the Extended Syntax Expansion check box in the DBL Options dialog box. (An example of an extended block ending is PROC...ENDSUBROUTINE for the SUBROUTINE keyword.) Click OK to exit the DBL Options dialog.
6. Click OK to exit the Extension Options dialog.

## Getting help with routine syntax

As you type text in the Workbench editor, you can display help for the current situation.

### Displaying context-sensitive routine help

To display a list of arguments for a subroutine, function, or method,

- Type the routine name, followed by an open parenthesis.

If the routine has more than one definition (for example, a routine with multiple subfunctions), the displayed syntax will say "1 of *n*," where *n* is the total number of definitions. You can cycle through the definitions by pressing ALT+comma (ALT+,) or clicking on the gray and black arrow buttons in the displayed routine help. (You can also press ALT+comma with the cursor anywhere in the routine name or syntax to display routine help in the first place.)

As you type each argument in the editor, it is boldfaced in the argument list so you can easily keep your place. Optional arguments are displayed in square brackets. (See figure 2-9.)

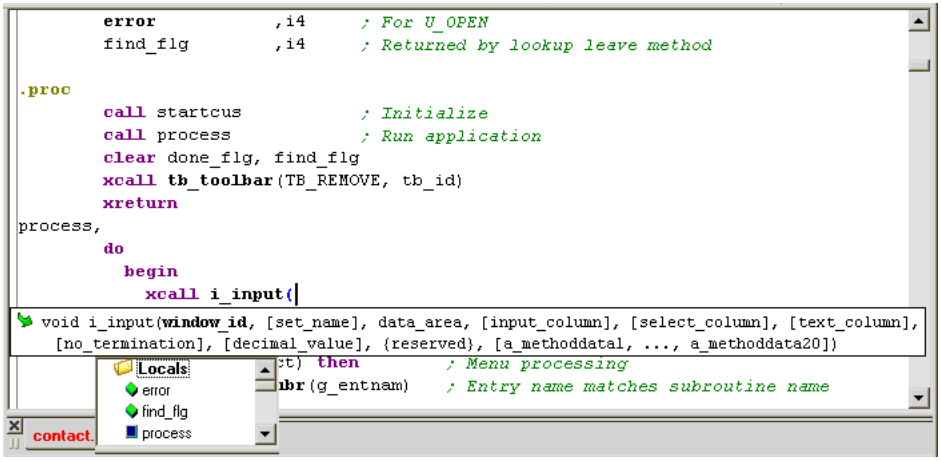



*Figure 2-9. Displaying context-sensitive help.*

This procedure works not only for Synergy Language and UI Toolkit routines but also for your own routines. Depending on how your project is set up, you may first need to set up a **.dbl**-specific tag file that references those routines. Refer to "Setting up a .dbl-specific tag file" on page 2-20 for more information.

### Displaying a list of valid operations for the current location

To list the valid variables, functions, macros, and so forth that can be used at a specific location, press ALT+period (ALT+.) at that location. If there is text before the cursor, the displayed list narrows down your options. For example, pressing ALT+period after "u_" provides a list of items that begin with "u_".

If you're specifying a record or group, when you press ALT+period after the record or group, a list of all members in that record or group automatically appears. If your cursor is at a field name or class variable followed by a period (optionally followed by more text or, in the case of a class variable, by the method name) and a right parenthesis, the Workbench editor displays the list of subfields for the field in question, matching text as described above. You can do this with repository structures as well, as long as you have .INCLUDEd the repository in your source code. (See "Including from a repository" in the Discussion for .INCLUDE in your *Synergy Language Reference* manual for more information.)

## Moving between routines in a single source file

Each named main routine, named function, named subroutine, macro, and .INCLUDE is represented on the project toolbar's Defs tab in a tree view. (See figure 2-10.)

To jump to a particular main routine, function, or subroutine in the source file, double-click on the routine name in the project toolbar.

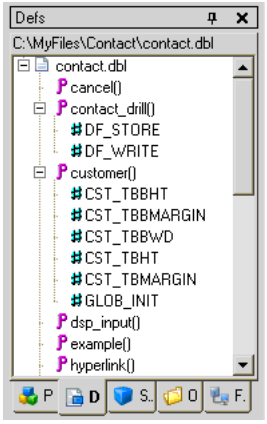



*Figure 2-10. The Defs tab.*

# Working with tag files

If you have multiple source files, you'll probably want to use the editor's tag file feature. A tag file lets you move between files using CTRL commands or get context-sensitive help for routines located outside the current file. A tag file is automatically created for each new project with the name of the project and the extension **.vtg**. You can add files to this file or create additional tag files if you desire. Once a tag file is created, it automatically updates in the background when you make edits.

We recommend that custom tag files be placed in directories outside the Synergy/DE directory tree. Any custom tag files should reside on your local hard drive.

The Symbol tab of the output toolbar displays tags for the word under the cursor. If only one tag is found, the source file containing the tag is displayed, and you can double-click on it to edit the file. Otherwise, the tag symbols are listed, and you can double-click on a tag symbol to go to that tag.

If the source file contains a .INCLUDE as the first line in the file, the Symbol tab does not display the definition of tag symbols in the file.

## Adding files to the workspace's tag database

1. From the Tools menu, select Tag Files (or click the Tag Files button on the Symbol tab). The Tag Files dialog box is displayed. (See figure 2-11.)

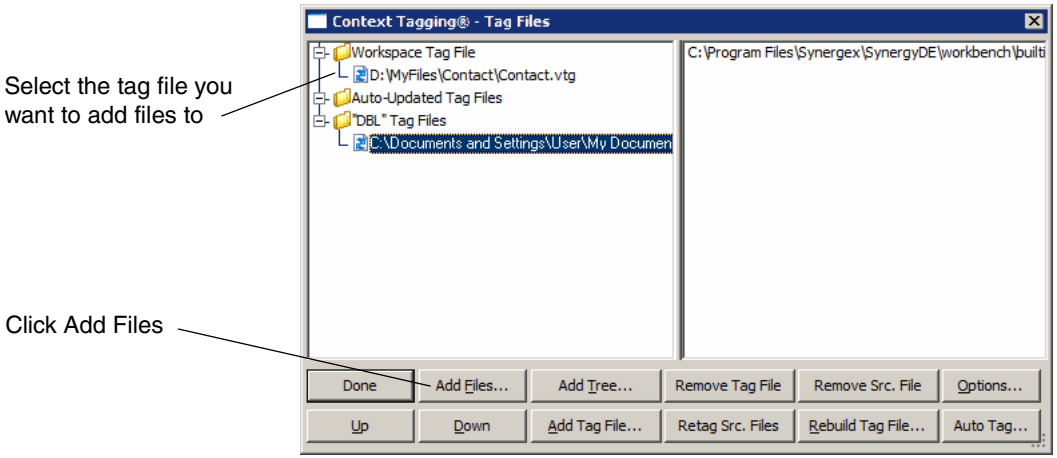



*Figure 2-11. Adding files to a tag file.*

2. Select the tag file you want to add files to and click the Add Files button. The Add Source Files dialog box is displayed.
3. In the File name field, type or browse for the name of a file you want to add and click the Open button. The file you select will be added to the list of files in your tag file.
4. Repeat steps 2 and 3 for as many files as you want to add.

## Setting up a .dbl-specific tag file

You might want to set up a separate, extension-specific tag file for files that, for one reason or another, you don't want to include in your project. For example, if you have a set of .INCLUDE or source files that you don't modify very often or that are used by other developers (such as your core routines), you probably want to maintain them outside of your project. Setting up separate tag files for such files lets you use the editor's tagging features for *all* symbols—not just the symbols in your project—including being able to view popup help for routines that aren't part of the current project.

To set up a **.dbl**-specific tag file,

1. From the Tools menu, select Tag Files (or click the Tag Files button on the Symbol tab).
2. Click the Add Tag File button.
3. In the Add Tag File dialog box, select **DBL** from the list and then click OK.
4. In the Add Tags Database dialog box, enter a name for the new tag file and click Open. The default filename is **tags.vtg**.
5. The Synergy/DE file extensions are listed in the Files field of the Add Tree dialog box. Click OK.
6. In the Tag Files dialog box, click Done when you are finished.

## Moving between files in your project

To jump to the definition of a routine or label in the source file (or vice versa),

1. Place the cursor on the invocation of the routine or label or on its definition.
2. Do one of the following:

| To move from | Press |
|---|---|
| The invocation of the routine or label to the definition | CTRL+period (CTRL+.) |
| The definition of the routine or label back to the invocation | CTRL+comma (CTRL+,) |

For example, placing the cursor over "sub1" in "xcall sub1" and pressing CTRL+period takes you to the definition of the subroutine **sub1**. Pressing CTRL+comma returns to the "xcall sub1" invocation.

## Displaying online Help

- Do one of the following:

| To find information about | Do this |
|---|---|
| The editor using a table of contents | Select Help > Contents |
| The editor using an index or full-text search | Select Help > Search |
| The editor using a list of frequently asked user questions and their answers | Select Help > Frequently Asked Questions |
| Other Synergy-specific aspects of Workbench | Select Synergy/DE > Workbench Help Topics |
| A specific dialog box or error message | Click the Help button in the dialog or message box |

# Generating Synergy Code Segments

Aliases and code templates are time-saving features of Workbench that generate common UI Toolkit code segments and method routines directly into your program.

## Using aliases

Workbench provides two types of aliases: directory (stored in the **alias.slk** file) and extension-specific (stored in the **dbl.als** file). Both are distributed in the workbench directory. When they are modified for the first time, they are copied to the SlickEdit configuration directory, and the modifications are made to that version of the file.

Using directory aliases means you don't have to type in long path names when prompted for a filename or directory. For more information about directory aliases, refer to the online Help. (See "Displaying online Help" on page 2-21.)

Extension-specific aliases are used for syntax and code expansion, enabling you to generate entire sections of code automatically. The remainder of this section describes how to use extension-specific aliases.

Before you begin using aliases in Workbench, follow the instructions in "Taking advantage of syntax expansion" on page 2-17 to turn on alias expansion.

### Invoking an alias

If the file you're editing has an extension associated with Synergy/DE, you can expand an alias as follows:

- Type the alias into the Workbench editor and press the spacebar or Enter.

Each alias displays a Parameter Entry dialog box that prompts you for such information as routine name, variable names, and so forth (see figure 2-12 for an example). After you enter the requested information, Workbench inserts the alias code into the current source code window, substituting the names you specified in the appropriate places.

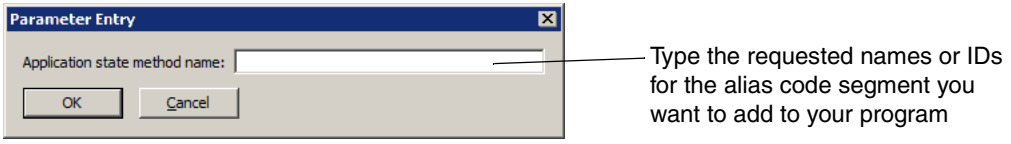

Parameter Entry
Application state method name:
OK
Cancel
Type the requested names or IDs for the alias code segment you want to add to your program

*Figure 2-12. Entering alias parameters.*

Workbench is distributed with aliases for the following code segments:

| Code segment | Alias |
|---|---|
| Application move event | synmeappmove |
| Application resize event | synmeappsize |
| Application state changed | synmeappstate |
| Application century method | synmecentury |
| User defined data - check field method | synmechkfld |
| Application close event | synmeclose |
| User defined data - display field method | synmedspfld |
| User defined data - edit display field method | synmeedtdsp |
| Application input override method | synmeentrst |
| User function key override method | synmefkey |
| User help method | synmehelp |
| User utilities method | synmeutils |
| Input field arrive method | synmiarrive |
| Input field change method | synmichange |
| Input field display method | synmidisplay |
| Input field drill method | synmidrill |
| Input field edit format method | synmieditfmt |
| Input field hyperlink method | synmihyper |
| Input field leave method | synmileave |
| List arrive method | synmlarrive |
| List double click method | synmldblclk |
| List leave method | synmlleave |
| List load method | synmlload |
| Tabset method | synmtab |

| Code segment | Alias |
|---|---|
| Window button method | synmwbutton |
| Window click event method | synmwclick |
| Window close event method | synmwclose |
| Window generic event method | synmwevent |
| Window maximize event method | synmwmax |
| Window minimize event method | synmwmin |
| Window move event method | synmwmove |
| Window restore event method | synmwrest |
| Window scroll event method | synmwscroll |
| Window resize event method | synmwsize |
| Synergy function | synfunc |
| Synergy subroutine | synsub |
| Input processing loop | syninp |
| List processing loop | synlist |
| Tabset processing loop | syntab |

## Customizing an alias

To modify either the code that is generated when you expand an alias or the parameters you are prompted for,

1. From the Tools menu, select Options > File Extension Setup, and select **dbl** in the Extension field.
2. From the General tab, click the Aliases button. The Alias Editor dialog box is displayed. (See figure 2-13.)
3. Edit as many aliases as you want to and then exit the dialog box.

   You can also use this dialog box to view which aliases are available.

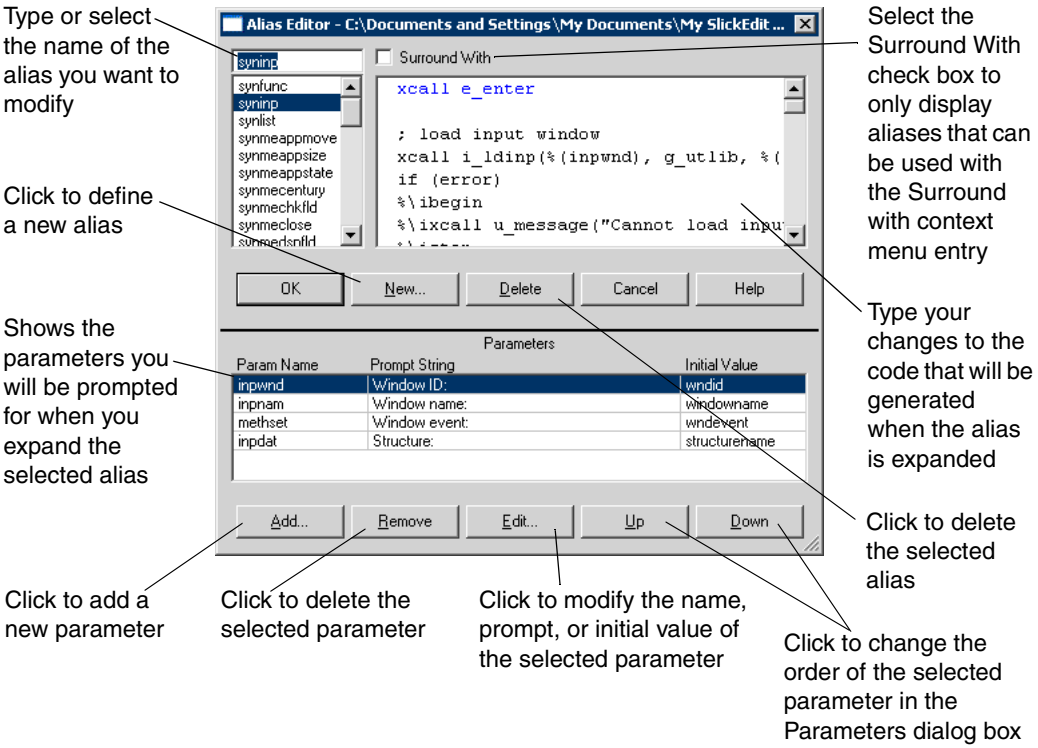



*Figure 2-13. Editing an alias.*

## Creating your own alias

1. From the Tools menu, select Options > File Extension Setup, and select **dbl** in the Extension field.
2. From the General tab, click the Aliases button. The Alias Editor dialog box is displayed. (See figure 2-13.)
3. Click the New button. The Enter New Alias Name dialog box is displayed.
4. Enter the name of the alias you want to create and then exit the dialog box.
5. In the editor field of the Alias Editor dialog box, type the code that will be generated when this alias is expanded.
6. Click the Add button. The Enter Alias Parameter dialog box is displayed.
7. Enter the desired information in each input field, as shown in figure 2-14, and then exit the dialog box.

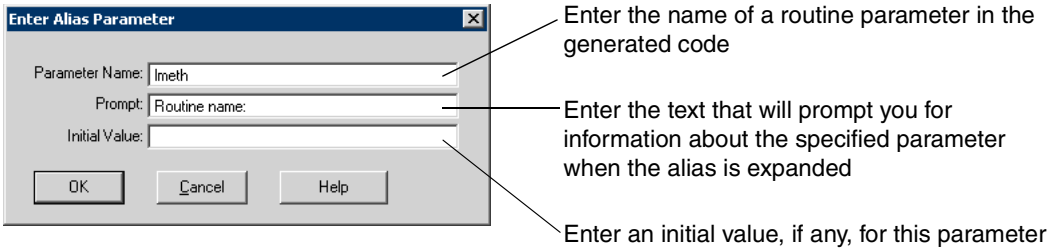

Enter Alias Parameter
Parameter Name:
lmeth
Prompt:
Routine name:
Initial Value:
OK
Cancel
Help
Enter the name of a routine parameter in the generated code
Enter the text that will prompt you for information about the specified parameter when the alias is expanded
Enter an initial value, if any, for this parameter

*Figure 2-14. Defining alias parameters.*

# Using code templates

Template files enable you to create generic versions of your Toolkit method routines. When the drilldown button for a method property of an input field or button is clicked in Composer or Repository, Workbench generates the appropriate method routine into the editor. You can then customize the routine as desired.

Template files have the extension **.tpl**.

To expand a template file,

1. In the Application window in Composer, select the input field or button object for which you want to generate a method routine.
2. In the Properties window, click the drilldown button for the desired method property. The Choose Method File dialog box is displayed in Workbench.
3. Type or browse for the name of the source file that contains or will contain the code for the method you selected in Composer. If the file does not exist, it will be created for you.

## Customizing template files

You can customize the template files that are distributed with Workbench. The ***.tpl** files are distributed to the workbench\wbsamples directory and then copied to the workbench directory if they don't already exist. If you are in a Terminal Services or shared configuration and you want all users to have a version that is different from the distributed version, edit the template files in the workbench directory. If only some of the users want to use a modified version, they can copy the desired ***.tpl** files to their SlickEdit configuration directory and then edit those files instead.

## Using tokens in template files

In the templates, you can use tokens to represent specific names. For example, structure and field tokens enable you to replace generic references to structures and input fields with references to actual structure and field names when generating methods for input windows. When Workbench inserts the template file's contents into the specified file, it replaces all instances of the tokens with the appropriate structure or field name.

Workbench is distributed with the following tokens:

| This token | Is replaced with | For this method |
|---|---|---|
| #STRUCTURE# | Structure name | Input field |
| #FIELD# | Input field name | Input field |
| #CURSOR# | The flashing cursor (so you can begin typing) | All |
| #ROUTINE# | Routine name | All |

If the field is a repository field, the #STRUCTURE# token is replaced with the repository structure name. If the field is not a repository field, the #STRUCTURE# token is replaced with the name of the structure the field belongs to in the input window.

To avoid potential conflict with the names of tokens we may add in the future, we recommend that if you create your own tokens, you either use a character other than the pound sign as the delimiter (for example, @*token*@), or you use a unique prefix (for example, your initials, as in #AK_*token*#).

# Analyzing Your Code

When you must restructure or otherwise modify a routine, it's important to be aware of possible ramifications to other pieces of code. Workbench provides some analysis tools to help you identify which routines are called by a specified routine and which routines call a specified routine.

## Viewing a call tree of external routines

The call tree utility lists all routines (across multiple source files) called by the chosen routine. You can also expand any routine in the call tree to see what routines are called, in turn, by that routine.

The primary objective of this utility is to provide insight into the logical structure of an unfamiliar software system. The generated list of called routines can be especially useful when you are converting legacy systems into components. Or if you want to expose a method in a component, generating a call tree of that method enables you to view all the methods called from that method, which can help you determine the scope of work needed to expose the method.

### Launching the call tree

A call tree can only be generated when a workspace is loaded.

1. From the Symbols tab of the project toolbar, right click on the routine for which you want to generate a call tree. (To locate the routine you want, expand the Workspace folder and then the Global Functions or Global Procedures subfolder, depending on whether you're looking for a function or a subroutine, respectively.)
2. Select Calls or uses from the context menu.

   A call tree is displayed for all methods under the selected method. (See figure 2-15.) The call tree continues searching through submethods until either there are no more submethods, the submethods loop back recursively to an existing method, or the call tree reaches a method that is defined outside of the project.

   To place your cursor at the beginning of a call in the Workbench editor, double-click on that call in the Symbol Uses/Calling Tree window.

### Determining what appears in the call tree

To toggle which items are included in the call tree,

1. Right-click in the Symbol Uses/Calling Tree window to display the context menu.
2. Highlight Quick filters in the context menu and then select the items you want to display.

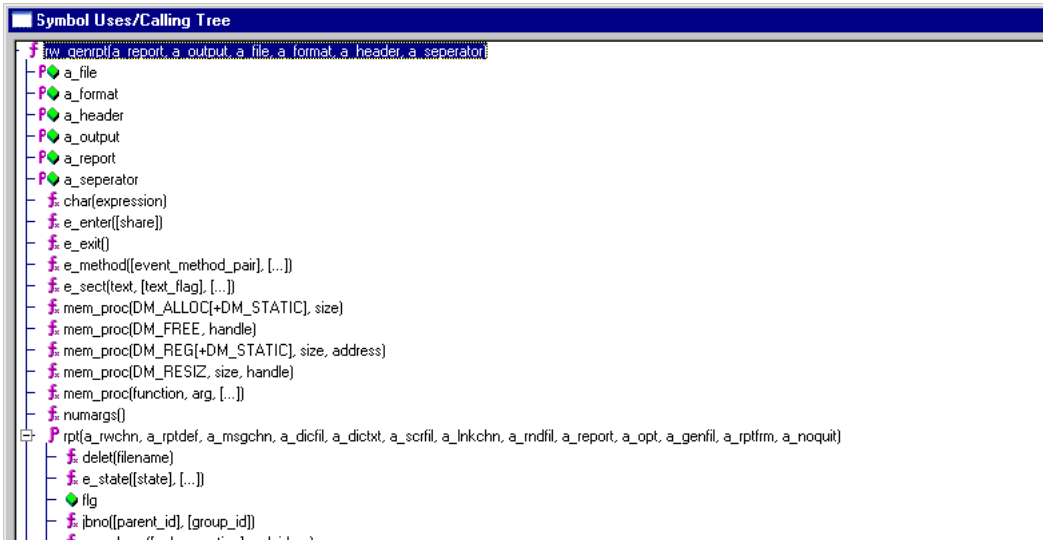



*Figure 2-15. Viewing a call tree.*

# Viewing where a method is called

When restructuring code for a component, you may want to know where a particular method is called, to help you identify which areas of code will be affected by changes to that routine.

## Launching the reference utility

References can only be generated when a workspace is loaded.

1. From the Symbols tab of the project toolbar, right click on the routine, class, class member, or local field for which you want to generate references. (To locate the item you want, expand the Workspace folder and then the appropriate subfolder, depending on what type of item you are looking for.)
2. Select References from the context menu.

   Workbench searches all files in all projects in the current workspace to create a list of calls to the selected routine or references to the selected class, class member, or local field. The call list is displayed in the References tab of the output toolbar. (See figure 2-16.)

   To locate a reference in the Workbench editor, double-click on the occurrence in the results window, or select the results window and press CTRL+G to move your cursor through the results window one line at a time. The appropriate file will be opened in the editor if it is not already open. The source line containing the reference is displayed in the preview window in the References tab if you highlight the occurrence in the results window.

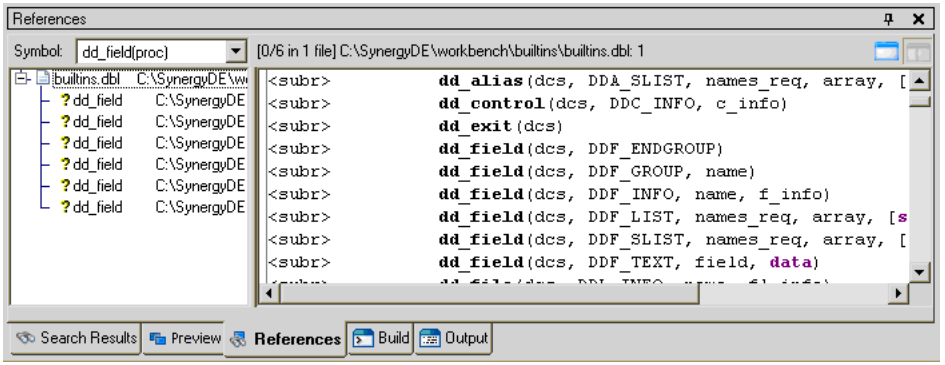



*Figure 2-16. Viewing routine references.*

### Another way to find references

1. From the Search menu, select Find.
2. Enter the name of the method or other item you want to search for and click the Files >> button.
3. In the Look in field, enter the files or group of files to be searched. If you click the dropdown button, you can select <Buffers> to search all open files, <Workspace> to search all files in the workspace, <Project> to search all files in the project, or the specified directory to search that directory. (Refer to the SlickEdit help for additional information about the Find dialog box or appending multiple search targets.)
4. Click OK to begin searching for all places where this method or other name appears in the selected files.

## Browsing an ActiveX control

To view all methods and fields within an ActiveX control,

1. Add the ActiveX control to a project if it's not associated with one already.
2. Select the Symbols tab on the project toolbar to display the Workbench class browser.
3. Locate the ActiveX control by expanding the Workspace folder and then the Packages/Namespaces subfolder.
4. Expand the ActiveX control or right-click on it to select an option from the context menu.

   You should be able to view all classes in the control, all methods in each class (including the parameters for each method), and all properties in each class.

# Compiling, Building, Running, and Debugging

After you've typed and edited your code, you can compile, build (or link), and run your project directly from Workbench by selecting the desired command from the Build menu or clicking the appropriate toolbar button. (See "Customizing and adding commands" on page 2-34 for instructions on changing your default commands.)

## Checking compilation errors

If Workbench encounters any compilation errors, it displays them in the Build tab of the output toolbar, which is below the edit window. You can move between compilation errors in the build window as follows:

| To move to | Do this |
|---|---|
| The next error | Press CTRL+SHIFT+DOWN ARROW |
| The previous error | Press CTRL+SHIFT+UP ARROW |
| A specific error | Click on that error |

As you select each error, that line of code is highlighted in the Workbench editor.

## Debugging a project

To compile and build in debug mode, set the SYN_DBG environment variable to **-d**, either in the **synergy.ini** file or using **syn_set** in the Open tab of the Project Properties dialog box (see "Setting environment variables in Workbench" on page 2-6 and "Customizing the way a project is opened" on page 2-33).

When you run the compiler and linker, the object and **.dbr** files are created with debugging information. See SYN_DBG in the "Environment Variables" chapter of *Environment Variables and System Options* for more information.

If you have customized your Compile and Build commands in the Project Properties dialog box (see "Customizing and adding commands" on page 2-34) and removed the text "%(SYN_DBG)," you will not be able to compile and build in debug mode as described above.

You can also execute a user-defined (or default) debug command by selecting Debug from the Build menu.

## Saving the current debugger state

The Synergy debugger enables you to save the current debugger state to a file. The debugger state includes break points, watch points, and option settings. By specifying the name of this debugger state file as the initialization file for the debugger, you can associate a set of debugger commands with a project and invoke those commands every time you restart the debugging session.

Use the debugger SAVE command to save the debugger settings to a file. (See the "Debugging Your Synergy Programs" chapter of *Synergy Language Tools* for details.)

To restore the saved debugger commands,

1. From the Project menu, select Project Properties and then click the Open tab.
2. In the editing area of the dialog, set the DBG_INIT environment variable to the name of the file that contains the saved debugger state.

   When you invoke the Synergy debugger, it will use the specified file to initialize itself.

# Customizing Your Development Environment

## Customizing the way a project is opened

To specify commands that will get executed when the project is opened,

1. From the Project menu, select Project Properties and then click the Open tab.
2. Enter the desired commands in the field shown in figure 2-17 and then exit the dialog box. (You do not need to close and reopen the project to make these changes take effect.)

Select the configuration you want to modify. Select All Configurations to change the settings for all of your configurations

Type any macro commands (one command per line) that you want executed when the project is opened

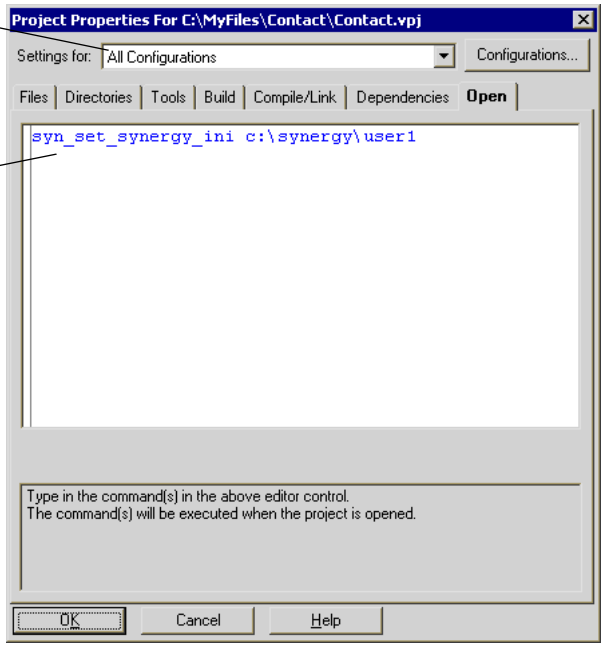



*Figure 2-17. Specifying project open commands.*

The commands that you specify in this dialog box are executed before any initialization settings in the [synergy] section of **synergy.ini** are loaded and set.

For details about specific commands, select Search from the Help menu and search for those commands.

TIP To save effort, we recommend setting SFWINIPATH instead of individual settings. (Refer to SFWINIPATH in the "Environment Variables" chapter of *Environment Variables and System Options* for details.)

# Customizing and adding commands

You can customize the commands for compiling, building, debugging, and executing for a project, for a project type, or for a file with an extension associated with Synergy/DE when no project is opened. For example, if you have a version control system, you can specify that the Build command on the Build menu is "build" or "make" or any other command you want it to be. Alternatively, you can specify a batch file that designates which files to build.

Workbench also enables you to create any number of tools to perform any desired commands. Two customizable tools (User 1 and User 2) are provided for you, or you can create a brand new tool. Note that any customized tool you create can only be activated from the menu and cannot be bound to a toolbar button or key press.

## Customizing commands for a project

You can modify the command lines for existing tools or add new tools that can be used by an entire project.

1. Open the project if it is not already open.
2. From the Project menu, select Project Properties.
3. In the Project Properties For *project* dialog box, select the Tools tab if it's not already displayed.
4. Enter the desired information in each input field (see figure 2-18) and then exit the dialog box.

   Once your project is set up, you can simply select Compile, Build, Debug, or Execute from the menu.

## Customizing commands for the Synergy/DE project type

You can modify the command lines for existing tools (Compile, Build, Debug, and so forth) or add new tools that Workbench will use when creating a Synergy/DE project.

1. Close any open workspaces and files.
2. From the Project menu, select New.
3. Click the Customize button.
4. In the Customize Packages dialog box, select the configuration you want to change the template for (for example, Synergy/DE) and click the Edit button.

Because Synergex reserves the right to override user settings in future versions of Workbench, we recommend you do not change the project packages for "Synergy/DE COM Component," "Synergy/DE Java Component," or "Synergy/DE .NET Component."

5. In the Project Package for *project* dialog box, select the Tools tab if it's not already displayed.
6. Enter the desired information in each input field, as shown in figure 2-18. Then exit the dialog box.

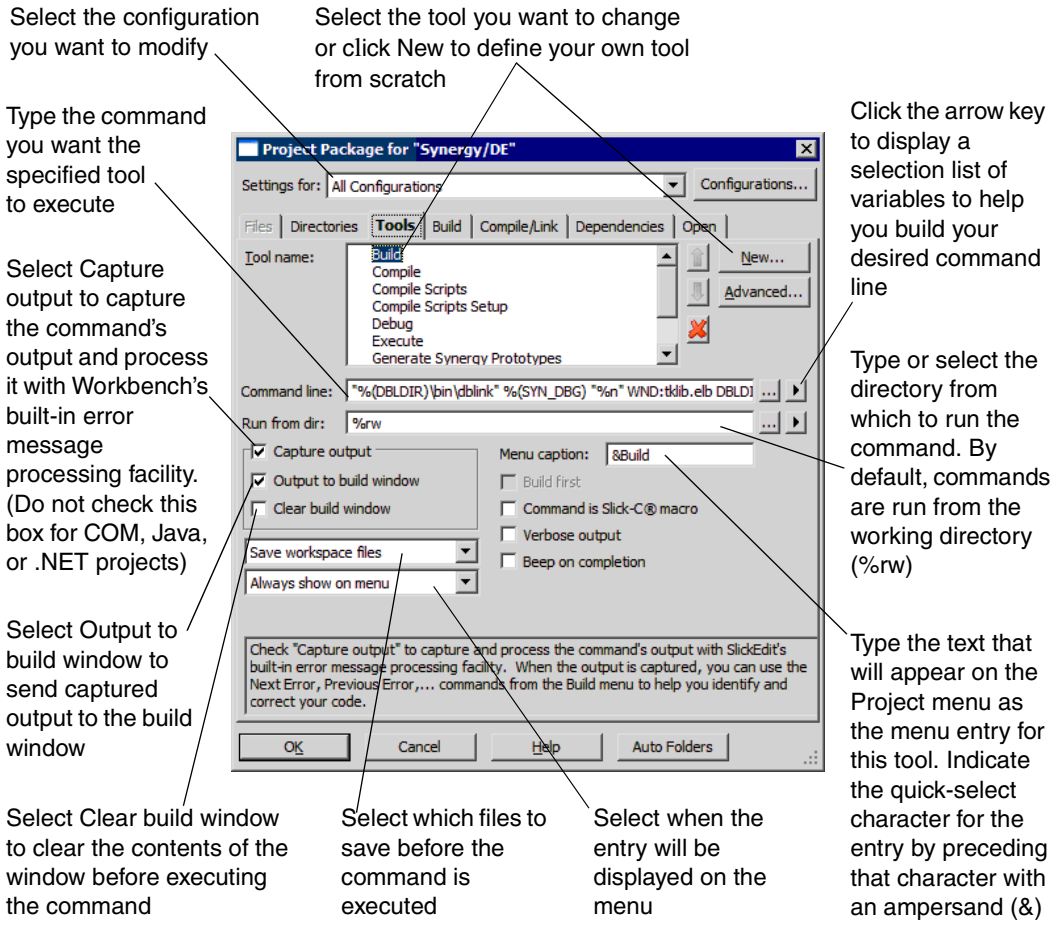

Select the configuration you want to modify
Select the tool you want to change or click New to define your own tool from scratch
Type the command you want the specified tool to execute
Click the arrow key to display a selection list of variables to help you build your desired command line
Select Capture output to capture the command's output and process it with Workbench's built-in error message processing facility. (Do not check this box for COM, Java, or .NET projects)
Type or select the directory from which to run the command. By default, commands are run from the working directory (%rw)
Select Output to build window to send captured output to the build window
Type the text that will appear on the Project menu as the menu entry for this tool. Indicate the quick-select character for the entry by preceding that character with an ampersand (&)
Select Clear build window to clear the contents of the window before executing the command
Select which files to save before the command is executed
Select when the entry will be displayed on the menu
Project Package for "Synergy/DE"
Settings for: All Configurations
Configurations...
Files
Directories
Tools
Build
Compile/Link
Dependencies
Open
Tool name:
Build
Compile
Compile Scripts
Compile Scripts Setup
Debug
Execute
Generate Synergy Prototypes
New...
Advanced...
Command line:
"%(DBLDIR)\bin\dblink" %(SYN_DBG) "%n" WND:tklib.elb DBLDI
Run from dir:
%rw
Capture output
Output to build window
Clear build window
Menu caption:
&Build
Build first
Command is Slick-C® macro
Verbose output
Beep on completion
Save workspace files
Always show on menu
Check "Capture output" to capture and process the command's output with SlickEdit's built-in error message processing facility. When the output is captured, you can use the Next Error, Previous Error,... commands from the Build menu to help you identify and correct your code.
OK
Cancel
Help
Auto Folders

*Figure 2-18. Customizing compiling, building, and running commands.*

All new projects of this type and configuration will inherit these tools when the project is created. If a project type has more than one configuration, you will need to customize each configuration individually.

## Customizing commands for a file being edited outside a Synergy/DE project

You can customize the commands for compiling, building, debugging, and executing on a global level when a **.dbl** file is opened with no open project or workspace.

1. Close any open workspaces and files.
2. From the Tools menu, select Options > File Extension Setup.

3. In the Extension field of the Extension Options dialog box, select **dbl**, and select the Advanced tab if it's not already displayed.
4. Click the Extension Specific Project button.
5. In the Project Properties For *extension* dialog box, select the Tools tab if it's not already displayed.
6. Enter the desired information in each input field (see figure 2-18), and then exit the dialog box.

   You can add your own commands using the New button, although they will not appear in the menu.

## Adding file extensions to Workbench

Workbench enables you to specify your own file extensions for source files, include files, and UI Toolkit script files in Synergy/DE projects. For example, unless you specify additional script extensions, only scripts with the extension **.wsc** will be recognized in Composer and Workbench. The extensions you specify will be added to the appropriate dialog boxes in Workbench.

To specify a new filename extension for a Synergy/DE project,

1. From the Synergy/DE menu, select Utilities > File Extensions.
2. Enter the desired information in each input field (see figure 2-19), and then exit the dialog box.

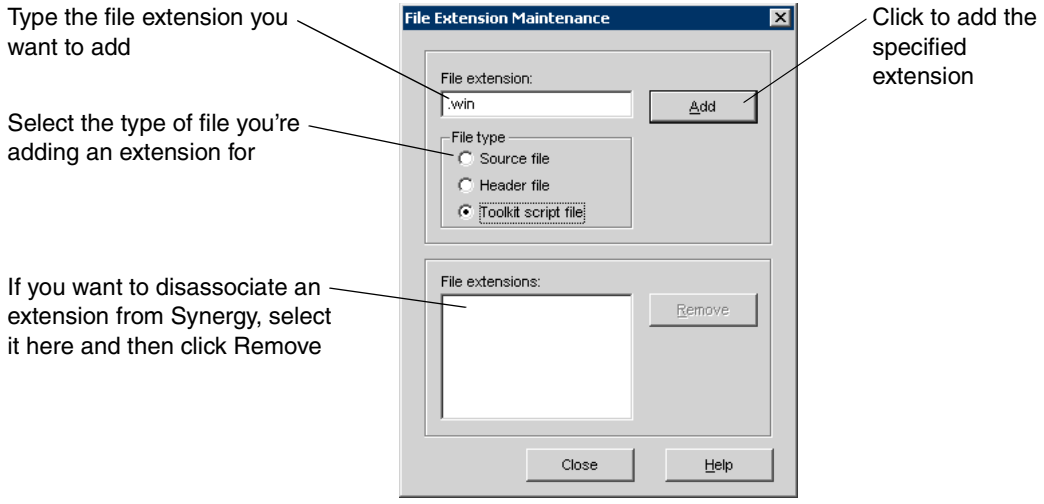



*Figure 2-19. Specifying different script extensions.*

3. If you've added a UI Toolkit script file extension and you want the default for a script file to be something other than **.wsc**, change the default extension in Composer by doing the following:
   - Run Composer.
   - From the Options menu, select Composer.
   - Click the File tab, and in the Default Extensions section, change the Script file value. Exit the dialog box.

## Customizing keyword color coding

Synergy keywords, compiler directives, and operators are automatically displayed in a different color from the rest of your code. To change the colors that are displayed,

1. From the Tools menu, select Options > Color Coding.
2. In the Lexer name field of the Color Coding Setup dialog box, select **dbl**.
3. Select the Keywords radio button to change the default keyword color, the Preprocessor radio button to change the default compiler directive color, or the Operators radio button to chanage the default operator color.
4. Click the Colors button.
5. In the Color Settings dialog box, click the desired text color in the Foreground field, or change the Red, Green, and Blue values as desired. You can also change the background color by selecting a color in the Background field.
6. Change the color of any other screen element by selecting the one you want to change in the Screen element field and performing step 5.
7. Exit both dialog boxes when you're finished customizing colors.

## Changing the tagging delay

You can customize how often tagging takes place by changing the number of seconds of idle time before tagging begins on a changed buffer. The default value is 3, which means open files are retagged immediately after you stop typing or using the mouse. Most people find this value too low. To change the tagging delay value,

1. From the Tools menu, select Options > Tagging Options.
2. Change the value in the Start tagging after seconds idle field.

   You can experiment with this value to find the one that works best for you. Depending on how fast you type, you may want to set this value to a number like 15 or 30. (In general, faster typists will probably prefer a higher value than slower typists.)

## Turning tagging off

To turn off automatic tagging,

1. From the Tools menu, select Options > Tagging Options.
2. In the Tagging Options dialog box, deselect Tag file on save and Background tag files.

   When Background tag files is not checked, Workbench only generates tags upon request.

Deselecting either of these options may cause all tagging to occur the moment you request it, which might cause Workbench to appear to hang.

## Interfacing with version control tools

Workbench provides support for several popular version control systems. To access these tools directly from Workbench,

1. From the Tools menu, select Version Control > Setup.
2. In the Version Control Setup dialog box, select the version control system of your choice (or create a new one using the Add button).
3. Click Setup and make any necessary changes to the version control commands, depending on your preferences or procedures.
4. Click OK.

   The Get and Put toolbar buttons are now enabled.

   For additional information, refer to the online Help system (see "Displaying online Help" on page 2-21) and select "Version Control" from the index.

## Understanding the .NET Framework version used by Workbench

When you create a .NET project, Workbench determines which version of the .NET Framework SDK is installed on your system and runs the **syn_set_sdkvars** macro, which runs **sdkvars.bat** to set up the environment for that version. For Synergy/DE 9.1.3, the only valid Framework version is 2.0. If you get a "Workbench cannot locate a valid version of the .NET Framework SDK" message, you need to install Framework SDK version 2.

Workbench adds the **syn_set_sdkvars** macro to the project open command (Project > Project Properties, and select the Open tab) for the current project, so that it is run automatically every time the project is opened, thus ensuring that the environment is set up properly each time.

# Copying customization settings

These instructions explain how to copy certain Workbench customization settings to another machine or to another user in a Terminal Services or shared configuration. The customizations that can be copied are aliases, template files, project templates, macros, and anything you customize by selecting any of the following entries from the Tools > Options menu:

- General
- File Extension Setup
- File Options
- Redefine Common Keys
- Visual C++ Setup

If the target machine has a brand new Workbench installation, enter the command "update_synergy" at the Workbench command line before beginning this procedure. Restart Workbench and change your editor configuration to match the machine whose configuration you are copying.

1. Copy the following files from the source SlickEdit configuration directory to the target SlickEdit configuration directory:

| File | Description |
|---|---|
| **alias.slk** | Directory aliases |
| **dbl.als** | Synergy-specific aliases |
| **diffmap.ini** | Diff dialog settings |
| **ftp.ini** | FTP setup |
| **oem.vlx** | Color |
| **project.vpe** | File extension command settings |
| **ubox.ini** | Comments |
| **uformat.ini** | Code formatting |
| **user.vlx** | User-defined color |
| **uservc.slk** | Version control setup |
| **usrprjtemplates.vpt**[a] | Project templates |

| File | Description |
|---|---|
| **vusrobjs.e** | User-defined dialog boxes and menus |
| **wbcfg.e** | Workbench configuration file |
| **wbkeys.e** | Workbench keymapping file |

a. The **usrprjtemplates.vpt** file is version specific and should therefore only be copied to a SlickEdit configuration directory of the same Synergy/DE version.

2. If any of your template files have been customized, refer to "Customizing template files" on page 2-26.
3. Run Workbench on the target machine.
4. Change the emulator configuration on the target machine to match the emulator settings on the machine from which you are copying.
5. At the Workbench command line, enter

   ```
   load_conf
   ```

6. Exit Workbench.

# Using Workbench for Non-Windows Development

Even though Workbench is only available on Windows, you can use it as your primary editor for all Synergy/DE-supported platforms via distributed computing. There are two options for accessing source files:

| Option | Pros | Cons |
|---|---|---|
| NFS (Network File System) based mapped drives | ▸ An NFS-based mapped drive is virtually identical to mapping a drive to a Windows server.<br>▸ Projects can be used.<br>▸ Composer can be used. | ▸ This option can be slower than FTP, although NFS can be optimized for better performance.<br>▸ NFS is not a very high-performance file server when used on OpenVMS. |
| FTP | ▸ This option is potentially faster, because you only transfer a file once and you edit it locally. | ▸ FTP is not suitable for systems using version control.<br>▸ If you have multiple developers, the potential exists for overwriting each other's changes.<br>▸ Projects cannot be used.<br>▸ Composer cannot be used. |

You will need to use *xf*Server to access your repository.

## Using NFS-based mapped drives

If you have problems with file loss or corruption, before using an NFS-mapped drive, we strongly recommend that you add "-s3" to the end of the command shortcut that runs Workbench. For example:

```
c:\synergyde\workbench\bin\vs.exe -s3
```

Using the NFS protocol to map a drive enables desktop users on a network to access remote source files and window scripts from UNIX and OpenVMS. If this option is too slow, you'll probably want to tweak the performance of your file server. Your NFS software can help you optimize performance.

We also recommend that you turn off automatic tagging in Workbench. (See "Turning tagging off" on page 2-38.)

# Using FTP

Workbench's built-in FTP functionality provides the ability to import source code directly into the editor from any other system on your network. Once you've set up the session protocols, Workbench's FTP tools enable you to edit remote code and compile it with the Windows compiler to check for errors. If you've set up your code to build on Windows, you can link it and perform test runs in the Workbench editor. You can then upload the modified source code to the original directory on the original platform simply by saving your work.

To set up FTP links between Windows and remote platforms, you must do the following:

1. Define your connection if it is not defined already.
2. Start a connection.
3. Open the file you want to edit and make your changes.
4. Transfer the file between your local machine and an FTP host.
5. Close the connection.

Basic instructions follow in the sections below. For additional information, refer to the *SlickEdit User's Manual*.

## Defining a connection

1. From the File menu, select FTP > Profile Manager. The FTP Profile Manager dialog box is displayed.
2. If you want to set up any firewall/proxy settings, log file options, or any other default settings, see "Customizing your FTP sessions" on page 2-44.
3. Click Add. The Add FTP Profile dialog box is displayed. (See figure 2-20 and figure 2-21.)
4. Enter the desired information in each field and click OK when you're finished.

   You can define as many sessions with full specifications as required. You can also create more than one session to the same host with each accessing a different remote directory with source files. In addition, you can specify the full path to the local work directory into which you want to download the code for each session.

## Starting a connection

1. In the FTP Profile Manager dialog box, select the connection you want to establish.
2. Click Connect.

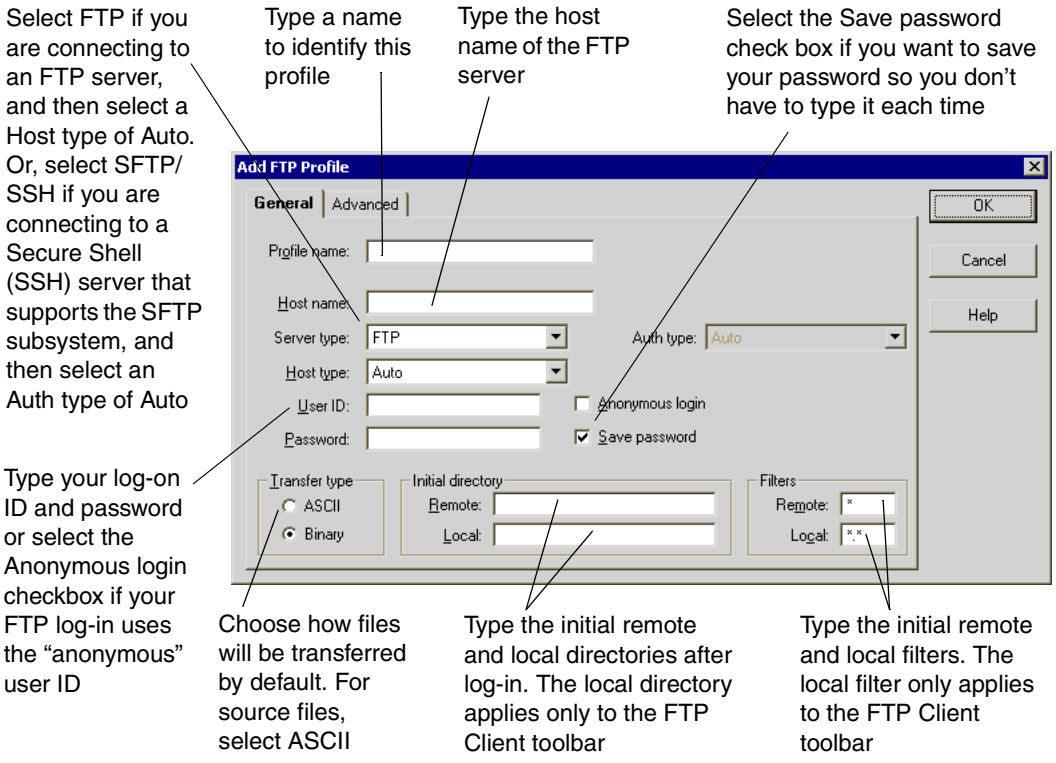

Select FTP if you are connecting to an FTP server, and then select a Host type of Auto. Or, select SFTP/ SSH if you are connecting to a Secure Shell (SSH) server that supports the SFTP subsystem, and then select an Auth type of Auto
Type a name to identify this profile
Type the host name of the FTP server
Select the Save password check box if you want to save your password so you don't have to type it each time
Add FTP Profile
General
Advanced
OK
Cancel
Help
Profile name:
Host name:
Server type:
FTP
Auth type:
Auto
Host type:
Auto
User ID:
Password:
Anonymous login
Save password
Transfer type
ASCII
Binary
Initial directory
Remote:
Local:
Filters
Remote:
*
Local:
*.*
Type your log-on ID and password or select the Anonymous login checkbox if your FTP log-in uses the "anonymous" user ID
Choose how files will be transferred by default. For source files, select ASCII
Type the initial remote and local directories after log-in. The local directory applies only to the FTP Client toolbar
Type the initial remote and local filters. The local filter only applies to the FTP Client toolbar

*Figure 2-20. Adding an FTP profile.*

## Opening a file for editing

1. After you have started a connection, click the Open tab on the Project toolbar. A listing of FTP files is displayed.
2. Double-click on a file to open it.
3. Edit the file as usual in the edit window.

## Transferring files between your local machine and an FTP host

The FTP Client dialog box enables you to open several FTP sessions at one time and then jump from session to session as desired.

1. From the File menu, select FTP > Client to display the FTP Client toolbar.
2. Right-click on the local or remote file listing to display a menu of operations from which to choose.

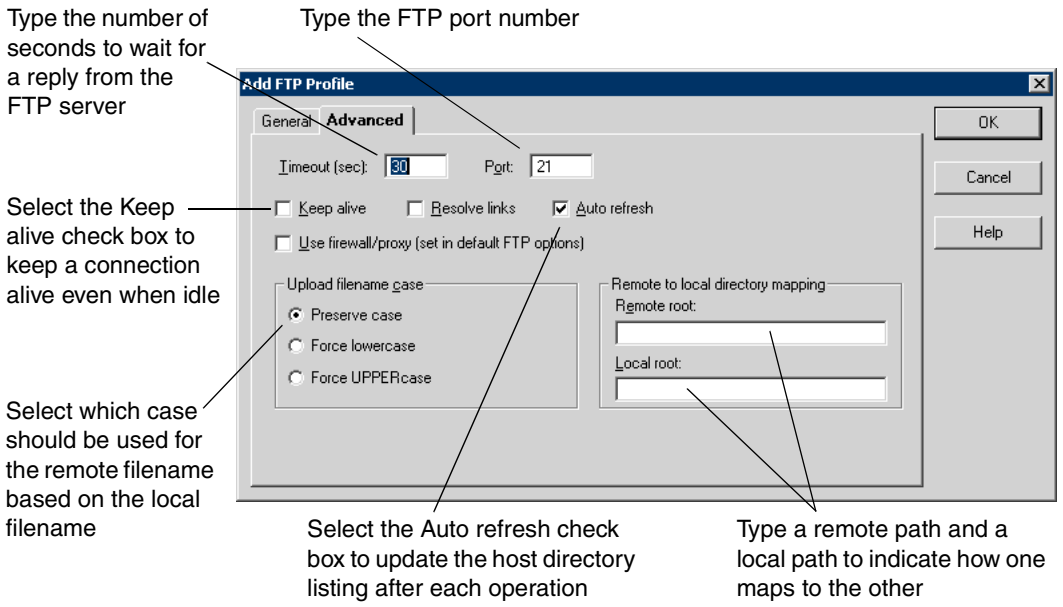

Type the number of seconds to wait for a reply from the FTP server
Type the FTP port number
Add FTP Profile
General
Advanced
OK
Cancel
Help
Timeout (sec):
30
Port:
21
Keep alive
Resolve links
Auto refresh
Use firewall/proxy (set in default FTP options)
Upload filename case
Preserve case
Force lowercase
Force UPPERcase
Remote to local directory mapping
Remote root:
Local root:
Select the Keep alive check box to keep a connection alive even when idle
Select which case should be used for the remote filename based on the local filename
Select the Auto refresh check box to update the host directory listing after each operation
Type a remote path and a local path to indicate how one maps to the other

*Figure 2-21. Adding an FTP profile (advanced settings).*

## Closing a connection

1. Open the FTP Profile Manager dialog box (File > FTP > Profile Manager) if it's not open already.
2. Select the connection you want to close in the Profiles field.
3. Click Close.

## Customizing your FTP sessions

You can optimize each session's protocols for the connecting remote host by customizing your FTP sessions in the FTP Options dialog box. From this dialog you can do the following:

- Specify a default password for anonymous log-ins
- Set default case sensitivity for the filenames you bring into Workbench from a UNIX system, so you can later upload those filenames back to the original system
- Specify whether you'll be prompted to upload each time code is saved in the Workbench editor (or that code should not be uploaded at all when it is saved)
- Specify how long Workbench will wait for a reply from the FTP server
- Set up firewall and proxy specifications if you are creating FTP sessions on secured networks
- Set up a session log to be created for all FTP sessions

To customize your FTP sessions,

1. If you're in the FTP Profile Manager dialog box, click Default Options. Otherwise, from the File menu select FTP > Default Options. The FTP Options dialog box is displayed.
2. Enter the desired information in each field of the appropriate tabs and click OK when you're finished.

# 3

# Setting Up Your Repository

This chapter introduces you to S/DE Repository and walks you through defining your data for several different uses. We've set up a small example for you to practice with, because we believe you will learn more quickly by actually performing the steps rather than just reading them.

For each of the instructional sections of this chapter, you will need to define a structure and its fields; we strongly recommend that you begin by following the instructions in the sections "Defining a structure" on page 3-3 and "Defining fields" on page 3-4, even if you don't need to define a record layout for use in an application.


### What Is Repository? 3-2

Describes what Repository is, how to run it, and how to access context-sensitive help.

### Defining a Record Layout for Use in an Application 3-3

Explains how to define the type and size characteristics of your data in Repository so it can be used by the Synergy compiler.

### Defining User Interface Characteristics 3-7

Explains how to define the visual and behavioral characteristics of your data in Repository so it can be used by S/DE UI Toolkit (including Composer).

### Defining Files for ReportWriter 3-15

Explains how to define additional visual and relational characteristics in Repository so your data can be used by Synergy/DE ReportWriter.

### Defining a Database Schema for xfODBC 3-21

Tells where to find instructions for using Repository with *xf*ODBC.

# What Is Repository?

Repository enables you to define the structure of your data in a centralized location, without redundancy. Repository can then supply this information to the Synergy compiler for processing application files, to Composer for building a user interface, and to ReportWriter for creating reports.

Because data is only defined once, changes need only be made once. For example, if you want to expand a field that appears throughout your application, one simple change to your repository does the trick.

## Starting Repository

To start Repository, do one of the following:

- From the Workbench toolbar, select Repository.
- From your SynergyDE folder in the Start menu, select Repository.
- At a command prompt, type

  ```
  dbr RPS:rps
  ```

  (If you are not in the directory that contains **dbr.exe**, DBLDIR\bin must be in your path.)

## Getting help

From any field in any input window in Repository, you can access context-sensitive help for that field. If you're not sure what information to enter in a field,

1. Place your cursor in the field.
2. Press F1 or click the Help button.

## Displaying a list of valid data for a field

Many fields in Repository also enable you to display a list of available selections. To display such a list,

- With your cursor in the field, select List Selections from the menu.

## See also

Your *Repository User's Guide*. This includes instructions for using all features of Repository, as well as information about Repository utilities, the Synergy Data Language (which describes the contents of a repository), and the subroutines that enable you to access repository data from within an application.

# Defining a Record Layout for Use in an Application

The most efficient way to define data fields is in one central location—a data repository—rather than in each of the programs that make up your application. The same definitions can then be accessed by multiple applications across multiple systems.

You will first need to define a structure and then define the data fields required by your application, including the data type and size of each field.

As an example for this chapter and the ones that follow, we've designed a simple application that maintains business contact information. It requires the following data fields:

```
company       ,a45
address       ,a45
city          ,a25
state         ,a2
zip           ,d5
salutation    ,a4
lname         ,a16
fname         ,a16
contact       ,a32
phone         ,d10
fax           ,d10
modem         ,d10
email         ,a47
notes         ,a47
type          ,a10
status        ,a10
sl            ,d1
tk            ,d1
rps           ,d1
pvcs          ,d1
odbc          ,d1
dte           ,d1
```

## Defining a structure

The first step in defining a record layout is to create a new structure. In Repository, the term *structure* refers to the entire record definition, or the combination of field and key characteristics.

To define a structure,

1. From the Modify menu, select Structures.
2. From the Structure Functions menu, select Add Structure.

   The Structure Definition input window is displayed. (See figure 3-1.)

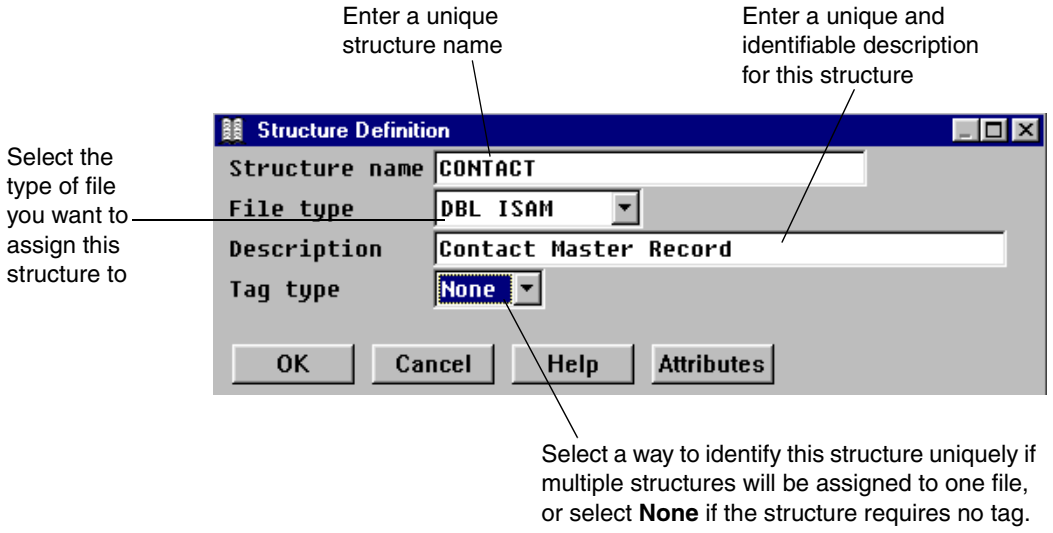

Enter a unique structure name
Enter a unique and identifiable description for this structure
Select the type of file you want to assign this structure to
Structure Definition
Structure name
CONTACT
File type
DBL ISAM
Description
Contact Master Record
Tag type
None
OK
Cancel
Help
Attributes
Select a way to identify this structure uniquely if multiple structures will be assigned to one file, or select **None** if the structure requires no tag.

*Figure 3-1. Defining a structure.*

3. Enter the desired information in each input field and then exit the input window.

## Defining fields

To define the necessary fields in your new structure, along with their data types and sizes,

1. With the new structure highlighted, select Edit Attributes from the Structure Functions menu.
2. From the Attributes menu, select Fields.
3. From the Field Functions menu, select Add Field.

   The Field Definition input window is displayed. (See figure 3-2.)

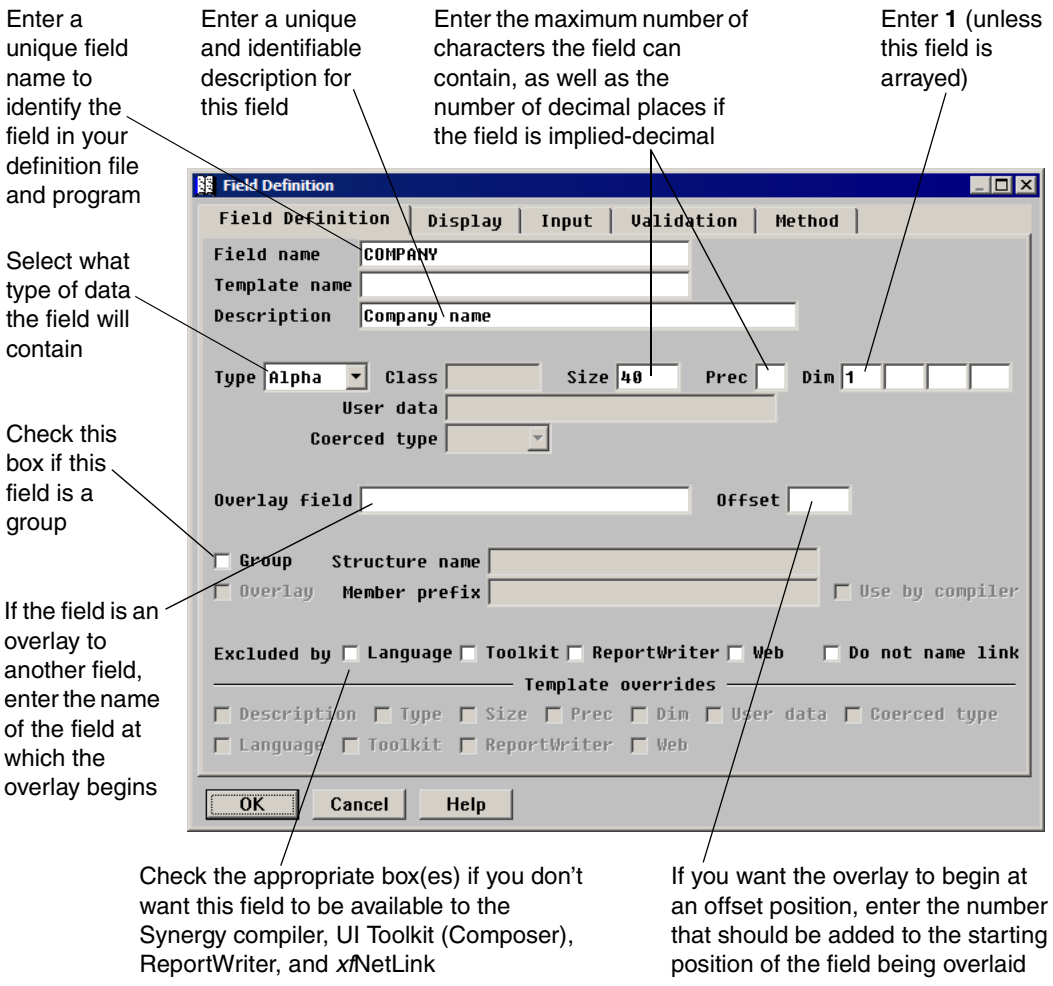



*Figure 3-2. Defining a field.*

**4.** Enter the desired information in each input field, using the example fields listed on page 3-3, and then click the OK button. (You can also define display, input, validation, and method information by moving to those tabs before clicking OK. These tabs are discussed in "Defining field attributes" on page 3-9.)

Make sure the Excluded from Language field is not checked; you will need to be able to access the fields you define later. You can ignore the Template overrides section for this example.

5. Repeat steps 3 and 4 for each field you want to define. We suggest you define all of the example fields except the **fax**, **modem**, and **notes** fields, which you can define by following the instructions in "Defining a field by copying an existing one" on page 3-6.

### Defining a field by copying an existing one

If a field you're defining is similar to an existing field, you can make a copy of the first field and then modify it to create a new field. For example, in our sample application, you can create the **fax** and **modem** fields by copying the **phone** field.

To copy a field,

1. Highlight the field you want to copy in the Field Definitions list.
2. From the Field Functions menu, select Copy Field.
3. Enter a unique name for the new field and modify the information in any of the remaining input fields as desired; then exit the input window.

### Reordering fields

The order in which the fields are listed determines the order in which they will exist within the structure. To move a field definition,

1. Highlight the field you want to move in the Field Definitions list.
2. From the Field Functions menu, select Reorder Fields.

   The highlighted field is now enclosed in square brackets ([ ]), indicating that it can be moved.
3. Use the UP ARROW and DOWN ARROW keys to move the bracketed field to another position in the list.
4. Select Reorder Fields again to get out of move mode.

## Saving your structure

When you attempt to exit the structure you're working on, Repository prompts you to save your changes.

To save your structure when exiting,

1. From the General menu, select Exit as many times as necessary. (For example, from the Field Definitions window, you'll need to select Exit twice.)
2. Select **Yes** when prompted to save your changes.

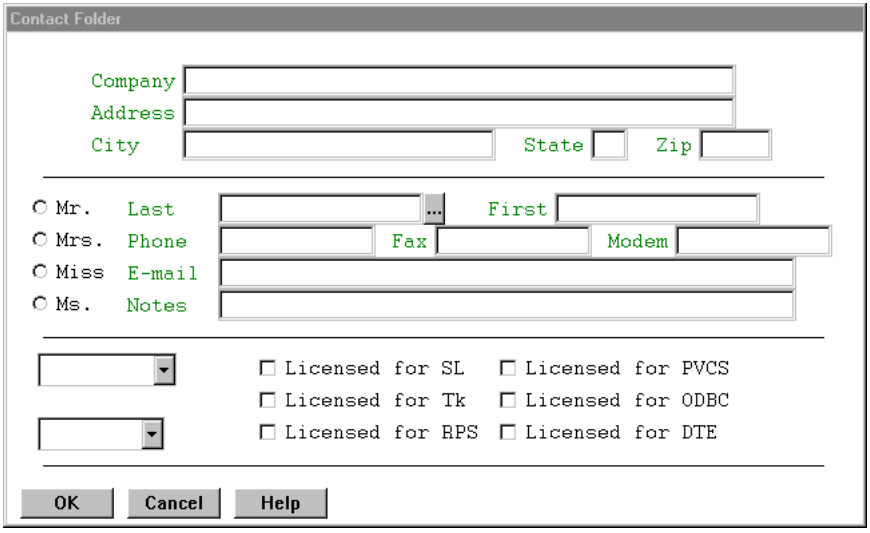
   You can either define another structure in the Structure Definitions list or you can select Exit again to return to Repository's main menu.

# Defining User Interface Characteristics

The data fields you define in Repository can be read into Composer as predefined input fields. Such centrally defined defaults simplify the process of supporting an application: when you modify a repository definition (and rebuild the applications that use that repository), that change is inherited by every object that refers to the original definition—regardless of which application the object is in. As a result, not only are the data definitions consistent from one application to another, but the user interface is consistent as well. In addition, many changes required to localize or customize an application can be accomplished by making changes in the repository, which enables you to customize your user interface more quickly, with less chance of error.

When you define fields for use in Composer, you'll want to define not only the fields themselves but also the fields' attributes and formats. Then, once you've added the fields to an input window in Composer, you can modify the fields and add features in the interface designer's visual, WYSIWYG environment.

Here's what the primary input window in our example application is going to look like:




## Defining a structure

Just like when you defined data fields for use by the Synergy compiler, the first step is defining a structure, as described on page 3-3.

Instead of defining another new structure here, you can use the structure you created on page 3-3 for our example user interface.

- From the Modify menu, select Structures.

# Defining how input is redisplayed in a field

When you define your fields' display attributes, you will have the option of selecting a predefined format to use when data is entered. For example, in our sample user interface, we'd probably want to assign a display format for the telephone number fields, Phone, Fax, and Modem.

So, to plan ahead, you can define two types of formats:

- Global
- Structure-specific

A global format can be used by a field definition in any structure. A structure-specific format is defined for a particular structure and can only be used by the fields in that structure.

To define a structure-specific format,

1. Highlight the structure for which you want to define a format in the Structure Definitions list.
2. From the Structure Functions menu, select Edit Attributes.
3. From the Attributes menu, select Formats.
4. From the Format Functions menu, select Add Format.

   The Format Definition input window is displayed. (See figure 3-3.)

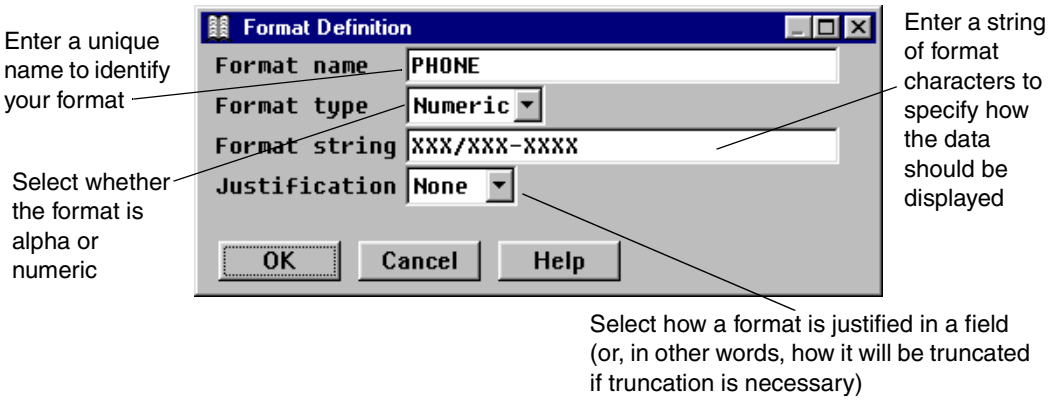

Enter a unique name to identify your format
Select whether the format is alpha or numeric
Format Definition
Format name
PHONE
Format type
Numeric
Format string
XXX/XXX-XXXX
Justification
None
OK
Cancel
Help
Enter a string of format characters to specify how the data should be displayed
Select how a format is justified in a field (or, in other words, how it will be truncated if truncation is necessary)

*Figure 3-3. Defining a format.*

5. Enter the desired information in each input field and then exit the input window.
6. Repeat steps 4 and 5 for each additional format you want to define.
7. From the General menu, select Exit.

If you want your format to be global instead of structure-specific, simply define it by selecting Formats from the Modify menu in Repository's main menu. Steps 4 through 7 are the same, regardless of the type of format you defined.

## Defining fields

The next step in creating a user interface is to define the required input fields. In our example, we're going to use the same fields we defined on page 3-4.

1. With your structure highlighted, select Edit Attributes from the Structure Functions menu. (If the Attributes menu is already present on your menu bar, go directly to step 2.)
2. From the Attributes menu, select Fields.
3. To view or edit the definition for an existing field, highlight that field and press ENTER.
4. Make sure the Excluded from Toolkit field is not checked for any of the fields; you will need to be able to access the fields you define later.

## Defining field attributes

Because we are designing a user interface, we need to define the appearance of each field on the screen as well as the field's behavior.

### Determining how the field will look

Repository refers to the field's appearance as "Display information." To specify display information for a field from the Field Definition input window,

1. On Windows, click the Display tab or press CTRL+TAB until the Display tab is displayed. On UNIX or OpenVMS, press TAB until the Display tab is displayed. (See figure 3-4.)
2. Enter the desired information in each input field and then exit the dialog box or move to a different tab in the dialog box to define additional information.

   (You can also get to the Display tab from the Field Definitions list. Highlight the field you want to modify, and select Edit Display Information from the Field Functions menu.)

### Determining how the field will behave

Repository breaks down the field's behavior into the categories "Input," "Validation," and "Methods." Input information determines how input must be entered and how it will be displayed and interpreted. Validation information defines what input is considered valid. Method information specifies any arrive, leave, drill, hyperlink, or change method subroutines associated with a field.

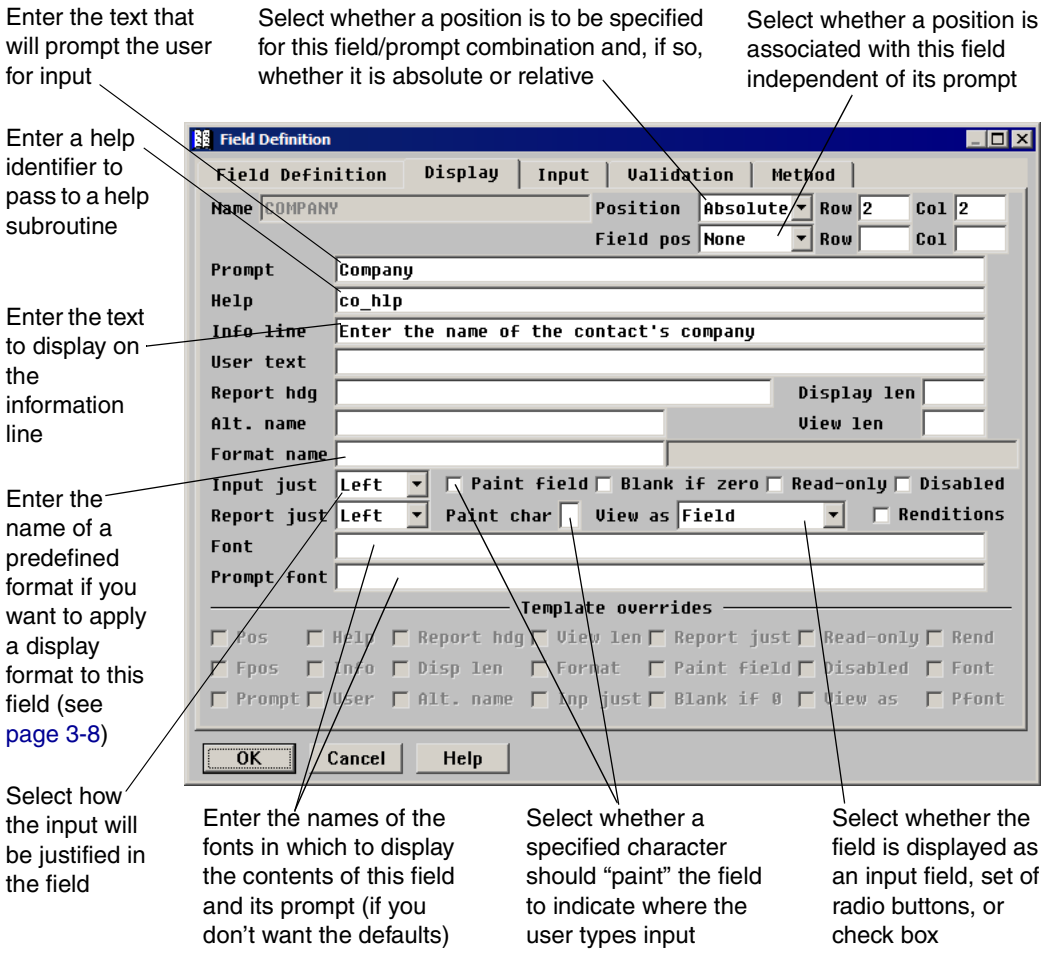



*Figure 3-4. Defining display information.*

### Specifying how input will be entered and displayed

To specify how input will be entered and displayed, do the following from the Field Definition input window:

1. On Windows, click the Input tab or press CTRL+TAB until the Input tab is displayed. On UNIX or OpenVMS, press TAB until the Input tab is displayed. (See figure 3-5.)
2. Enter the desired information in each input field and then exit the dialog box or move to a different tab in the dialog box to define additional information.

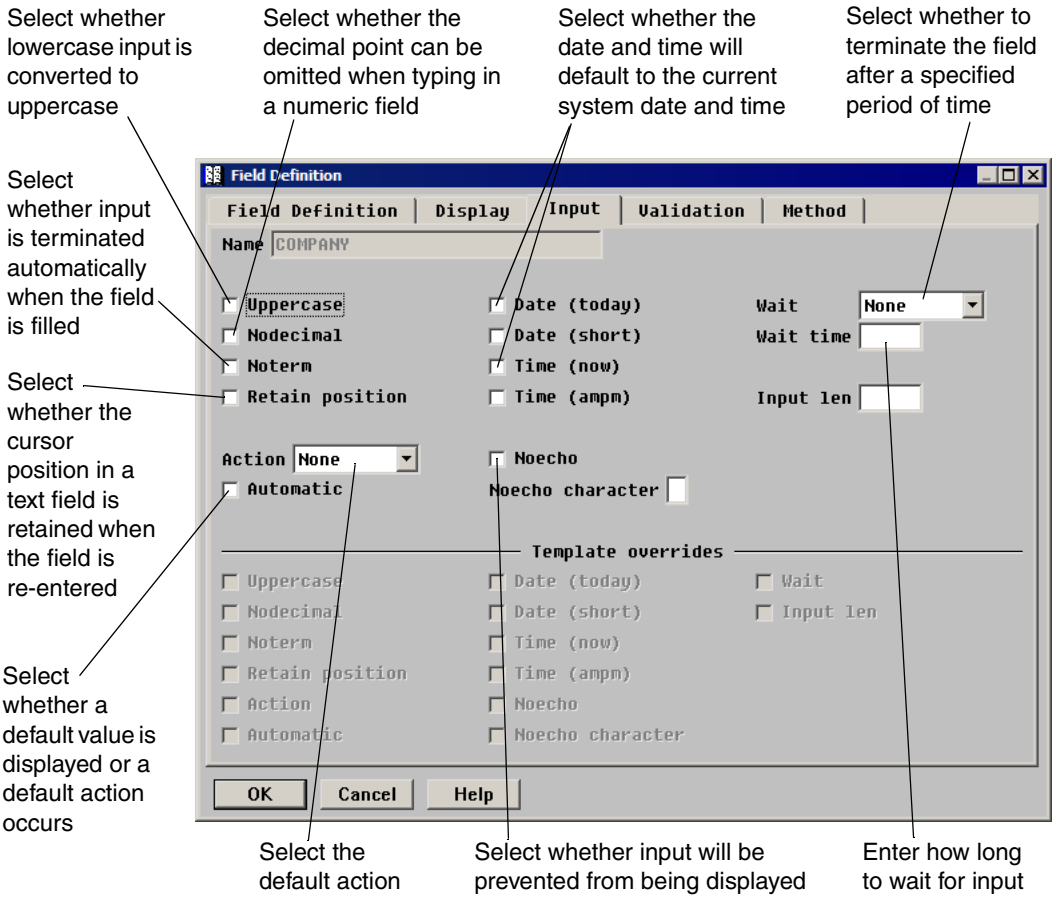



*Figure 3-5. Defining input information.*

(You can also get to the Input tab from the Field Definitions list. Highlight the field you want to modify, and select Edit Input Information from the Field Functions menu.)

### Specifying how input will be validated

To specify how input will be validated, do the following from the Field Definition input window:

1. On Windows, click the Validation tab or press CTRL+TAB until the Validation tab is displayed. On UNIX or OpenVMS, press TAB until the Validation tab is displayed. (See figure 3-6.)
2. Enter the desired information in each input field and then exit the input window or move to a different tab in the dialog box to define additional information.

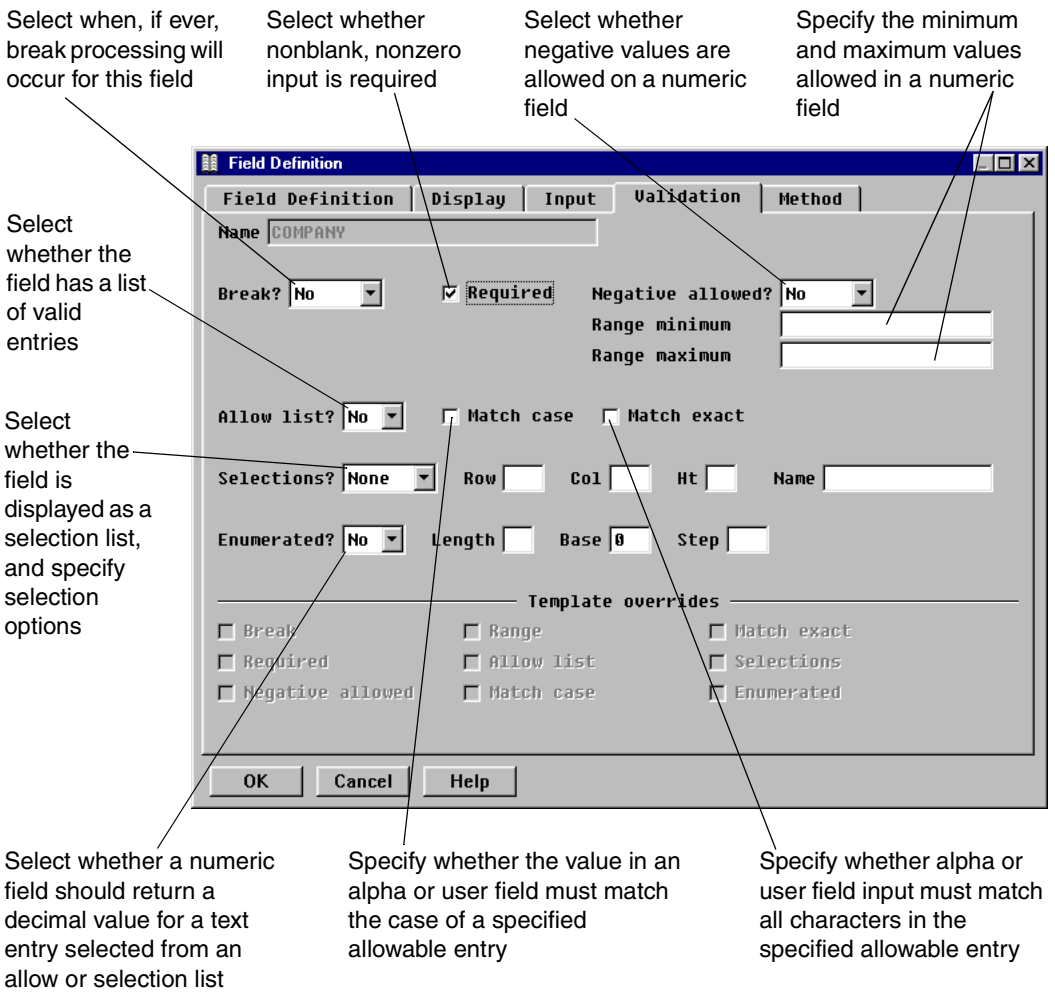



*Figure 3-6. Defining validation information.*

(You can also get to the Validation tab from the Field Definitions list. Highlight the field you want to modify, and select Edit Validation Information from the Field Functions menu.)

### Specifying method subroutines for a field

You can assign the following types of method subroutines to an input field:

- Arrive method
- Leave method

- Drill method
- Hyperlink method
- Display method
- Edit format method
- Change method

An *arrive method* subroutine is called to perform special processing before an input field is processed, while a *leave method* subroutine performs additional processing after an input field is processed (for example, validating the input).

If a *drill method* subroutine is specified, a drilldown button is displayed after the field in Windows environments. The subroutine is called when the user clicks the drilldown button (in Windows) or selects a drilldown menu entry (in Windows or non-Windows environments). A drill method is primarily used to look up additional information or display an input window.

A *hyperlink method* subroutine is called when the user clicks on an input field prompt. When a prompt is associated with a hyperlink method, that prompt is a hyperlink and the foreground color of the prompt is green.

A *change method* function is called after a field is validated by the UI Toolkit I_INPUT subroutine. It performs additional processing after an input field is changed (for example, validating radio button and check box fields).

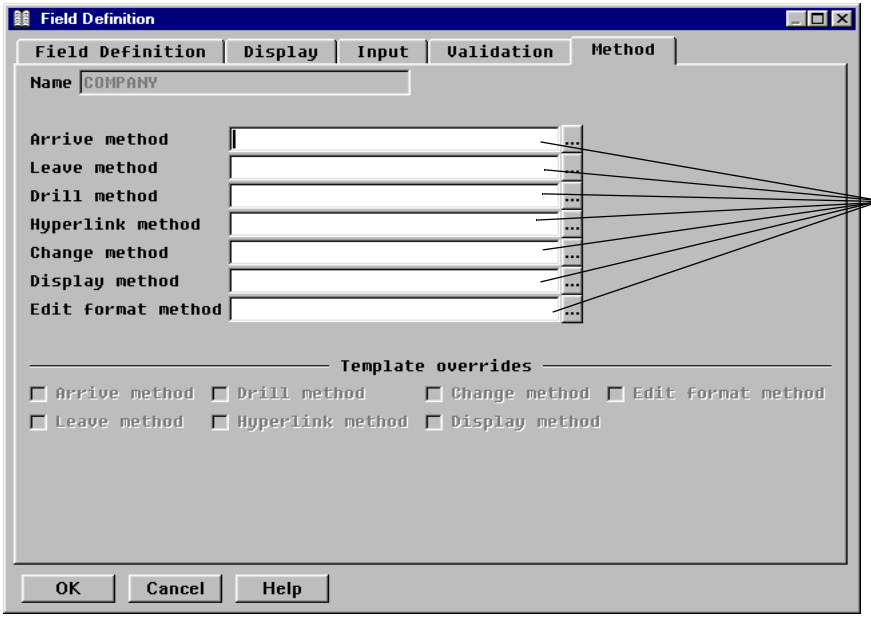
A *display method* function is called before UI Toolkit displays the contents of an input field. Your display method can modify the display format of the input field, as well as its color and attributes, thus enabling you to override Toolkit's display format for the field.

An *edit format* method function is called before UI Toolkit has formatted the contents of an input field to be edited. Your edit format method can modify the edit format of the input field, as well as its color and attributes, thus enabling you to override Toolkit's edit format for the field.

To specify method subroutines for a field, do the following from the Field Definition input window:

1. On Windows, click the Method tab or press CTRL+TAB until the Method tab is displayed. On UNIX or OpenVMS, press TAB until the Method tab is displayed. (See figure 3-7.)
2. Enter the desired information in each input field and then exit the input window or move to a different tab in the dialog box to define additional information.

   (You can also get to the Method tab from the Field Definitions list. Highlight the field you want to modify, and select Edit Method Information from the Field Functions menu.)




Enter the name of a method subroutine that you've written or plan to write. Click the drilldown button to open the Choose Method File dialog in Workbench, where you can create or modify the subroutine in the text editor.

*Figure 3-7. Defining method information.*

# Defining Files for ReportWriter

Repository seamlessly supplies information to Synergy/DE ReportWriter for creating reports. When you define fields for use in ReportWriter, you need to define not only the fields themselves but also the way the field will be used and displayed in a report, the keys that determine the relationships between files and how those keys will be linked with keys from other structures, and which files use which structures.

## Defining a structure

As when you were defining data fields for use by the Synergy compiler, the first step is defining a structure.

1. From the Modify menu, select Structures.
2. Select the structure you defined on page 3-3.

## Defining fields

The next step in creating a user interface is to define the required input fields. In our example, we're going to use the same fields we defined on page 3-4.

1. With your structure highlighted, select Edit Attributes from the Structure Functions menu. (If the Attributes menu is already present on your menu bar, go directly to step 2.)
2. From the Attributes menu, select Fields.
3. To view or edit the definition for an existing field, highlight that field and press ENTER.
4. Make sure the Excluded by ReportWriter field is not checked for any of the fields; you will need to be able to access the fields you define later.

## Determining how each field will be displayed in a report

In addition to defining the field's general appearance, which we did on page 3-9, "Display information" contains field data that affects how the field is used and displayed in a report.

To specify display information for a field,

1. Highlight the field you want to modify in the Field Definitions list.
2. From the Field Functions menu, select Edit Display Information.

   The Display Information input window is displayed. (See figure 3-8.)

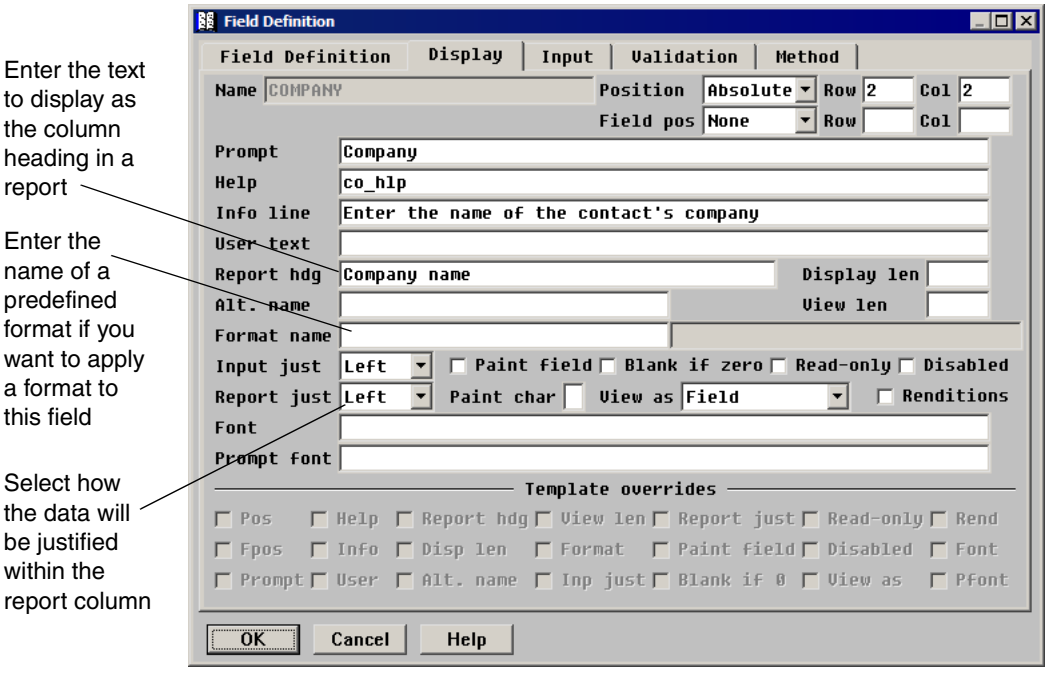



*Figure 3-8. Defining how a field will be displayed in a report.*

3. Enter the desired information in the Report hdg, Format name, and Report just input fields and then exit the input window. (See page 3-16 for instructions on predefining a format.)
4. Repeat steps 1 through 3 for each field you want to define.

## Defining a format

When you specify display information for a field, you can specify a predefined format to use with that field in a report generated by ReportWriter.

If you defined a format for an input field on page 3-8, you can use that same format here. Otherwise, follow the instructions on page 3-8 to define a display format.

## Defining a key to your record

A *key* is the portion of the data record that identifies the record and is used to access it. For example, if you define a company name field as a key with no duplicates, a record can be uniquely identified based on the company name. The keys you define in Repository also determine what relationships you can set up between database files, which in turn determine what additional information ReportWriter can access outside of a particular file.

The first key you define (or the first key in the list) is the primary key. A primary key must be an access key.

To define a key,

1. Highlight the structure you want to define a key for in the Structure Definitions list.
2. From the Structure Functions menu, select Edit Attributes.
3. From the Attributes menu, select Keys.
4. From the Key Functions menu, select Add Key.

   The Key Definition input window is displayed. (See figure 3-9.)

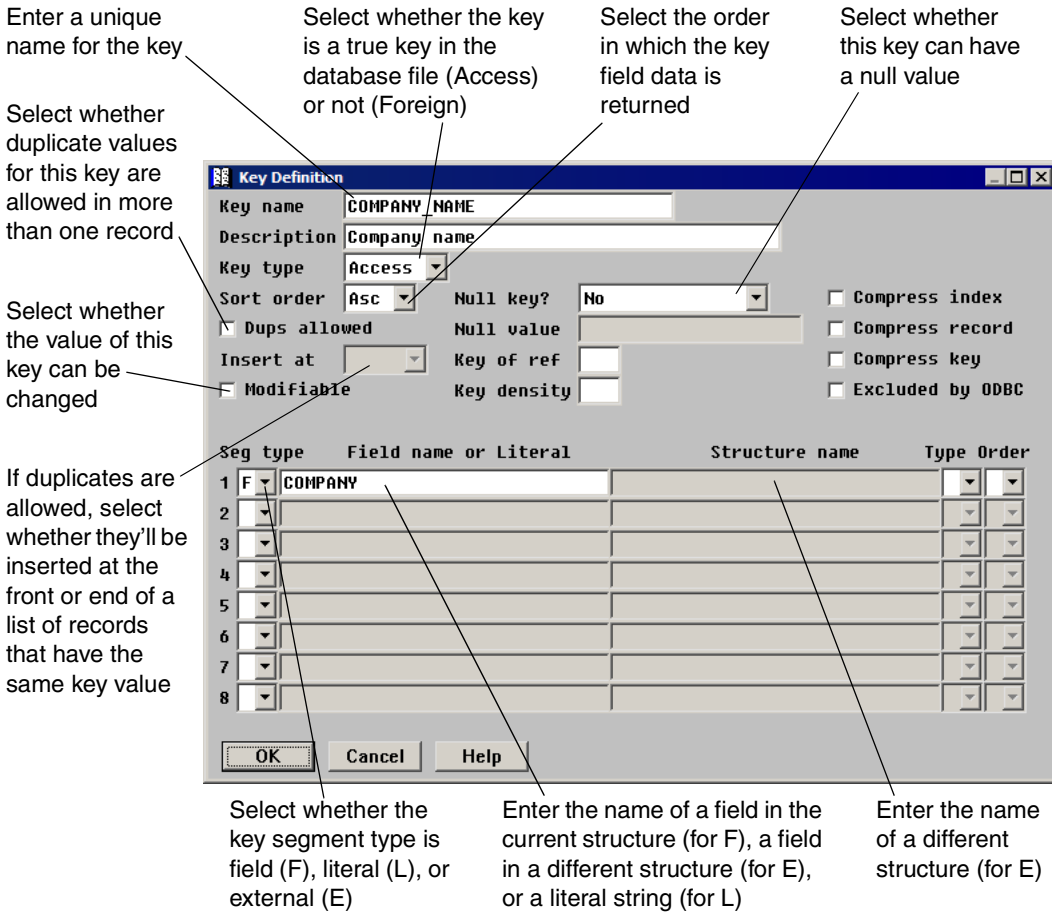

Enter a unique name for the key
Select whether the key is a true key in the database file (Access) or not (Foreign)
Select the order in which the key field data is returned
Select whether this key can have a null value
Select whether duplicate values for this key are allowed in more than one record
Select whether the value of this key can be changed
If duplicates are allowed, select whether they'll be inserted at the front or end of a list of records that have the same key value
Key Definition
Key name
COMPANY NAME
Description
Company name
Key type
Access
Sort order
Asc
Null key?
No
Compress index
Dups allowed
Null value
Compress record
Insert at
Key of ref
Compress key
Modifiable
Key density
Excluded by ODBC
Seg type
Field name or Literal
Structure name
Type
Order
1
F
COMPANY
2
3
4
5
6
7
8
OK
Cancel
Help
Select whether the key segment type is field (F), literal (L), or external (E)
Enter the name of a field in the current structure (for F), a field in a different structure (for E), or a literal string (for L)
Enter the name of a different structure (for E)

*Figure 3-9. Defining a key.*

5. Enter the desired information in each input field and then exit the input window.
6. Repeat steps 4 and 5 for each key you want to define.
7. When you're finished defining keys, select Exit from the General menu.

## Defining a relation between two structures

To enable ReportWriter to cross-reference data between files, you must create *relations* between structures. A relation enables you to link the keys of one structure with the keys of another structure. For example, if you relate a company name key in a contact management file with a company name key in a sales transaction file, ReportWriter can access sales information for each company when you create a report of your business contacts. We recommend that you set up every relation you can think of that a ReportWriter end-user might possibly need.

To define a relation,

1. Highlight the structure you want to define a relation for in the Structure Definitions list. (If the Attributes menu is already present on your menu bar, skip steps 1 and 2 and go directly to step 3.)
2. From the Structure Functions menu, select Edit Attributes.
3. From the Attributes menu, select Relations.
4. From the Relation Functions menu, select Add Relation.

   The Relation Definition input window is displayed. (See figure 3-10.)

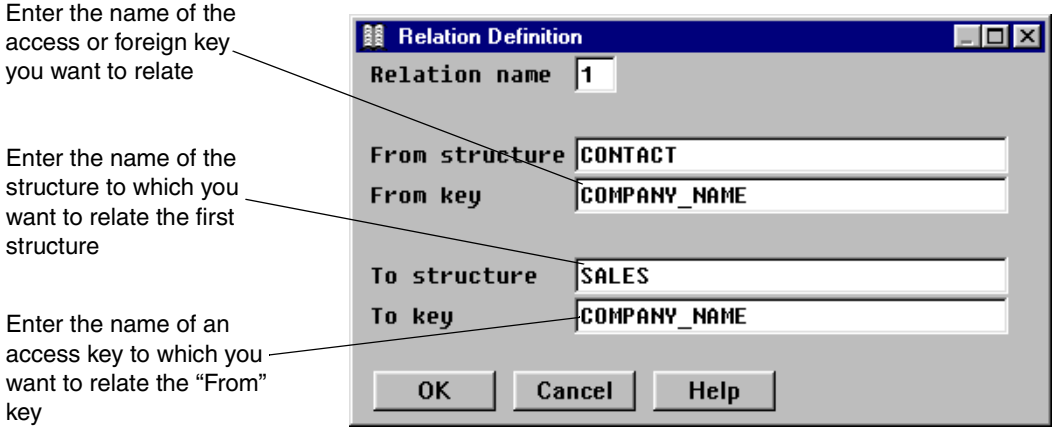



*Figure 3-10. Defining a relation.*

5. Enter the desired information in each input field and then exit the input window.
6. Repeat steps 4 and 5 for each relation you want to define.

# Defining a file

Defining your database files through Repository enables you to access those files from ReportWriter. Once you define a file you can assign one or more structures to it, thus specifying which structures can be used to access that file.

To define a file,

1. From the Modify menu in Repository's main menu, select Files.
2. From the File Functions menu, select Add File.

   The File Definition input window is displayed. (See figure 3-11.)

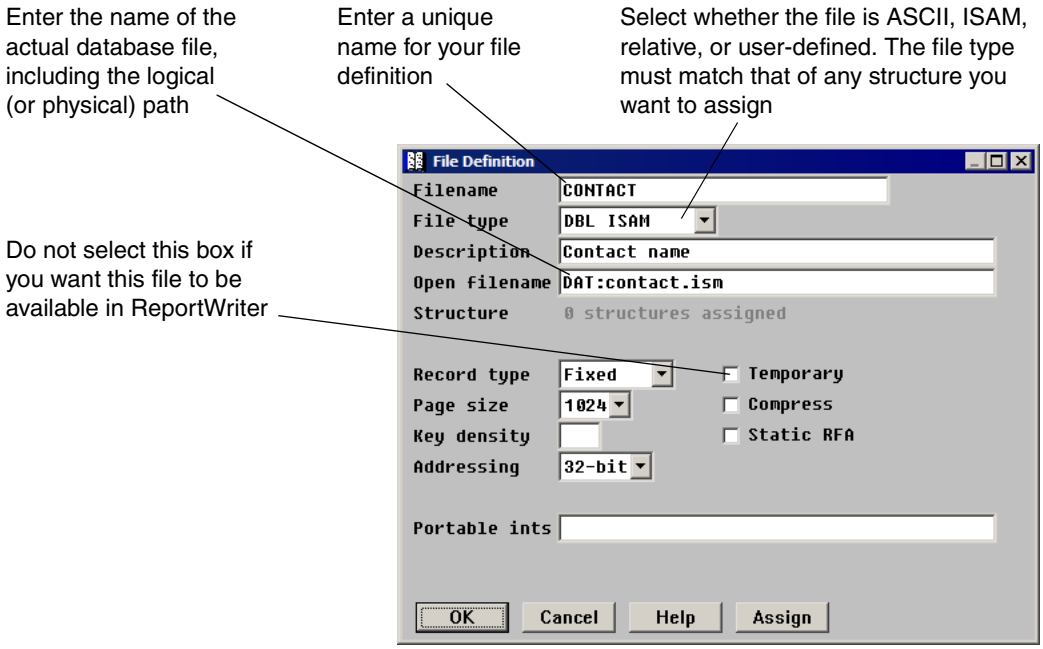



*Figure 3-11. Defining a file.*

3. Enter the desired information in each input field and then exit the input window.
4. Assign a structure to this file by following the instructions in "Assigning a structure to a file" on page 3-20.

## Assigning a structure to a file

You must assign a structure to a file in order for the file to be able to use that structure. You can only assign a structure that has the same file type as the file to which you're assigning.

1. From the File Definitions list, select Assign Structures from the File Functions menu.
2. From the Structure Functions menu, select Add Structure.
3. Enter the name of a structure to assign to the file.
4. Repeat steps 2 and 3 for each structure you want to assign.
5. From the General menu, select Exit twice to return to Repository's main menu.

# Defining a Database Schema for *xf*ODBC

A component of *xf*ODBC, the **dbcreate** utility, uses repository data definitions to generate a system catalog, which prepares a Synergy database for ODBC access. A system catalog describes a Synergy database in a way that the *xf*ODBC driver can understand. You can run **dbcreate** either from the command line or from within the *xf*ODBC Database Administrator (DBA).

When you define data for use by *xf*ODBC, your repository should contain a complete set of structures, tags, fields, and file information; templates for similar fields; and well-chosen keys and relations. See "Setting Up a Repository" in the "Preliminary Steps" chapter of the *xfODBC User's Guide* for detailed instructions.

# 4

# Designing Your User Interface

Now that you've set up your repository, you can start designing your user interface. This chapter introduces you to S/DE Composer, UI Toolkit's visual user interface designer for Windows environments. We will take you through the steps of designing an input window and several of the objects it can contain.


### Important Terminology 4-2

Defines several of the terms used in this chapter.

### What Is Composer? 4-3

Describes what Composer is, what it looks like, and how to run it.

### Using Composer 4-8

Walks you through the design of an input window that contains predefined input fields imported from your repository, radio buttons, check boxes, selection list fields, a drilldown to a second input window, lines, standard input fields, and command buttons. This section also describes how to save and close a script file, close a project, and compile your script.

# Important Terminology

## What is an object?

In this manual, when we refer to an *object* we mean any window, list class, menu column, window field, input field, selection window entry, menu entry, text, line, box, or button. Each object is an independently designed element that has certain characteristics and values and exhibits certain behavior.

## What is a window?

The term *window* refers to the window you're designing and encompasses general windows, input windows, and selection windows.

- A *general window* usually conveys information, such as error or informational messages.
- An *input window* contains one or more defined input fields into which the user can enter data.
- A *selection window* contains a group of predefined selections from which the user can choose.

## What are attributes and properties?

An *attribute* is a characteristic of an object, and a *property* is the value of an attribute. For example, color is an attribute, and red is a property.

## What is a script file?

A *script file* is an editable text file that contains special script commands to define and control user interface objects and their characteristics. When you design your user interface in Composer, the script commands are generated into a script file for you. Once your script file is created, it must be compiled.

The advantage of using script files is that your user interface objects are defined outside of your programs. You can modify object properties without changing or recompiling your Synergy Language subroutines.

## What is a project?

Composer organizes your script files in *projects*. A project is a collection of one or more related script files (for example, all of the script files for a particular application).

# What Is Composer?

S/DE Composer is a powerful user interface designer that enables you to create the elements of your interface graphically using the mouse and keyboard on Windows systems. You can easily move, resize, and change the characteristics of your interface objects in a WYSIWYG environment. Then, after you interactively design your general windows and input windows, Composer will generate the appropriate window scripts.

## Starting Composer

To start Composer, do one of the following:

- From the Workbench toolbar, select Composer.
- From your SynergyDE folder in the Start menu, select Composer.
- At a command prompt, type

  ```
  dbr SYNBIN:composer
  ```

  If you are not in the directory that contains **dbr.exe**, DBLDIR\bin must be in your path. Likewise, the SYNBIN environment variable must be in your path, and it must be set to the directory in which Composer is installed.

## What's on your Composer screen?

Composer consists of four windows:

- The Control Bar window
- The Application window
- The Properties window
- The Object Manager window

### The Control Bar

The Control Bar manages the entire Composer application. It contains Composer's menu bar and toolbars. To exit Composer, close the Control Bar window.

Composer has two toolbars, separated by a line:

- The Standard toolbar
- The Object toolbar

The Standard toolbar gives you quick mouse access to the most common project management, object editing, and window and tool access functions. It has two tab sets: Composer and Tools.

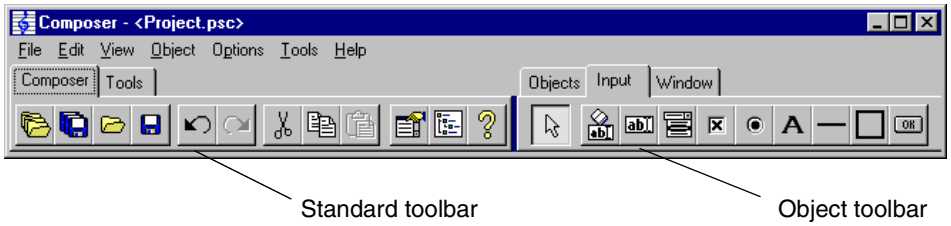



*Figure 4-1. The Control Bar window.*

The Object toolbar enables you to create and manipulate user interface objects quickly. It consists of an arrow pointer button and a tabbed toolbar. The arrow pointer button enables you to select objects in the Application window. The toolbar has three tab sets: Objects, Input, and Window.

To display the name of a toolbar button, hold your mouse pointer over the button for two seconds.

## The Application window

The Application window provides an area in which you can design your user interface. This design area contains a grid that represents cell-based rows and columns. When you move or size user interface objects, they snap to the grid. Objects created in the design area also contain grids, so that objects created within those objects will snap to row/column boundaries as well. You can turn off the grids if you need to (for example, when you want to print your screen).

The title bar for the Application window contains the name of the current script file.

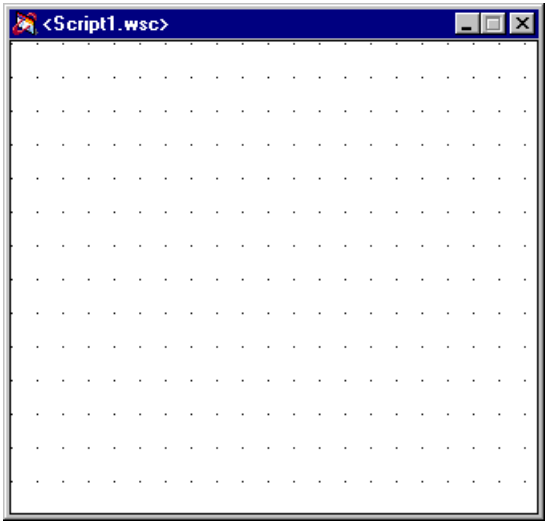



*Figure 4-2. The Application window.*

## The Properties window

The Properties window contains a list of design object attributes, followed by the property, or value, for each attribute. The attributes and properties displayed always belong to the object that's currently selected in the Application window. If no object is selected in the Application window, the properties displayed are those of the active script file.

Some attributes have subattributes and must be expanded before you can view or modify the subattributes. Such attributes are preceded by a plus sign (+). When the attribute is expanded, it is preceded by a minus sign (–). To expand or collapse an attribute, double-click on it.

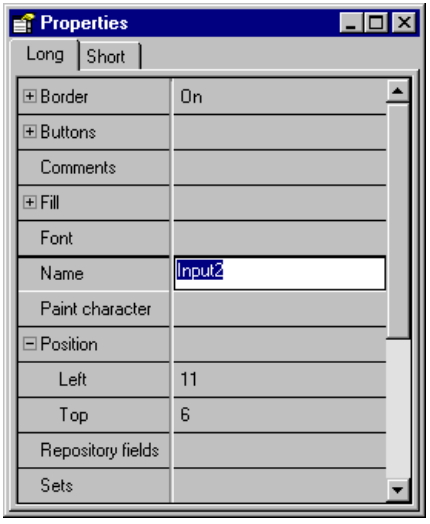



*Figure 4-3. The Properties window.*

## The Object Manager

The Object Manager window displays a visual representation of the project file, script file(s), and user interface object hierarchy in the following order:

- Project file
- Script file
- Containing object
- Contained object

From Object Manager, you can move, copy, delete, or rename objects.

The Object Manager window has its own menu bar, which contains functions available in Object Manager. It also has a status bar at the bottom to indicate the activity being performed.

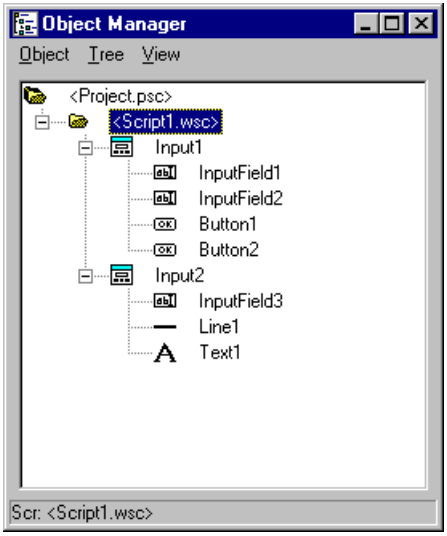



*Figure 4-4. The Object Manager window.*

The order in which the objects are listed is the order in which they are generated when the script file is written or updated. Files or objects that can be expanded are preceded by a plus sign (+). Expanded files or objects are preceded by a minus sign (–). To expand or collapse a file or object, double-click on it. Script files or objects that have not been fully compiled display the letter U (for "unprocessed").

## Using Help

Composer has an extensive online Help system to give you instructions for just about anything you'll need to do. If you can't find the information you need in this guide, you can search the online Help for additional options and procedures:

- From the Help menu, select Composer Help Topics, or click the Help Topics button on the toolbar.

From the Help Topics screen (see figure 4-5), you have three ways to search for information:

- Click the Contents tab to browse through topics by category.

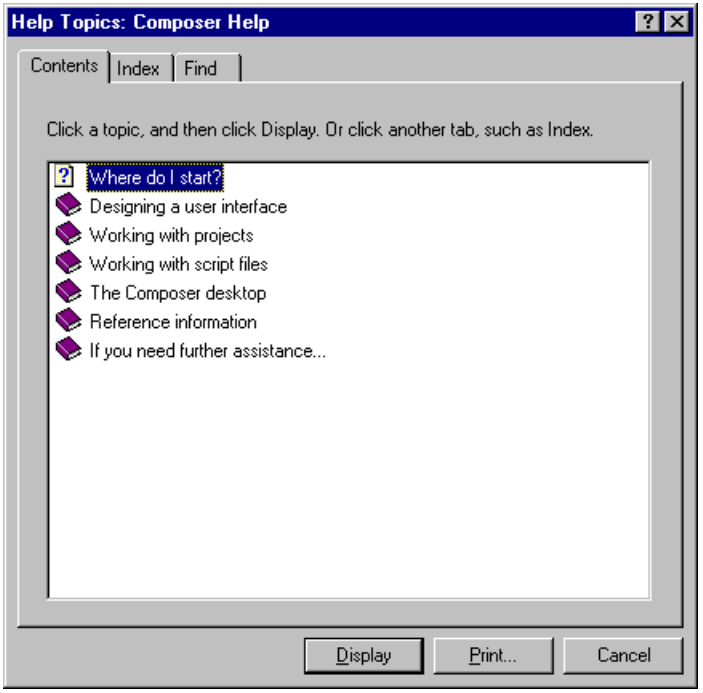



*Figure 4-5. Using Composer Help.*

- Click the Index tab to see a list of index entries. Either type the word you're looking for or scroll through the list.
- Click the Find tab to do a full-text search of words or phrases that may be in a Help topic.

Composer also provides help that is specific to the currently selected or highlighted item. To display context-sensitive help for a Composer window, dialog box, property, or object, do one of the following:

- Select the item and press F1. (From a dialog box, you can also click the Help button.)
- Point to the item with the arrow pointer, click the right mouse button, and select Context Help from the displayed context menu.

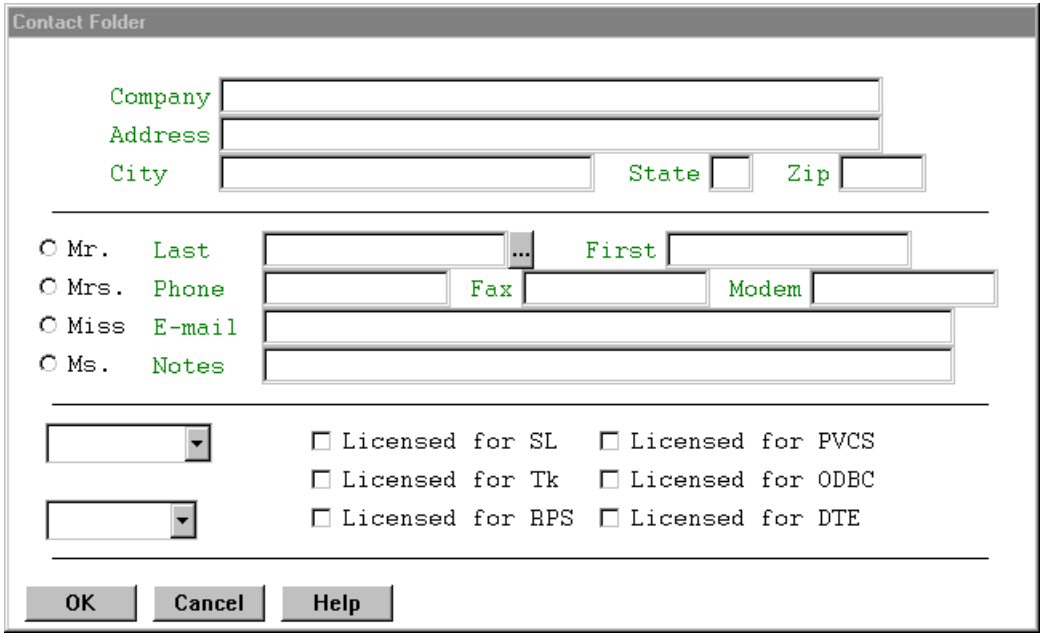
To display help for a menu command,

- Pull down the menu, highlight the command (without selecting it), and press F1.

# Using Composer

While this manual is not a tutorial, we believe that the quickest way to get you designing with Composer is for you to set up a sample user interface screen. As an exercise, you can use the same application for which you set up a repository in chapter 3, “Setting Up Your Repository.”

Here’s the primary input window you’re going to design:




We recommend that you run Composer and actually create this input window (or one of your own) as you read through the procedures below. Don’t be afraid to experiment and make mistakes: you can always click the Undo (or Redo) button if you change your mind.

## Designing an input window using repository fields

### Creating an input window

When you create any object in Composer, you can either let Composer place the object in a default location, or you can specifically place it where you want it to go. These instructions tell you how to create an input window of the size and in the location you desire:

1. Click the Input window button on the toolbar.

2. Click and drag on the desired location in the Application window until the window is large enough to hold the fields shown in the screen above.

   To create an input window of a default size in a default location, simply double-click the Input window toolbar button.

### Adding a window title

To specify a title to be displayed in the window's title bar,

1. Select the window by clicking on it in the Application window.
2. Click the Title property in the Properties window and enter the name of the title you desire (Contact Folder, in our example).

## Adding predefined repository fields

You can add input fields you've defined in S/DE Repository to the input window you just created.

1. Select the input window in the Application window if it's not already selected.
2. Click the Repository field button on the toolbar.

   The Repository Fields dialog box is displayed. Because this is the first repository field you're adding, you need to add a repository structure before you can place any fields. (If you were adding a second or any subsequent repository fields, you could skip steps 3 through 7 below.)

3. Click the Select button.

   The Select Repository Structure dialog box is displayed. (See figure 4-6.)

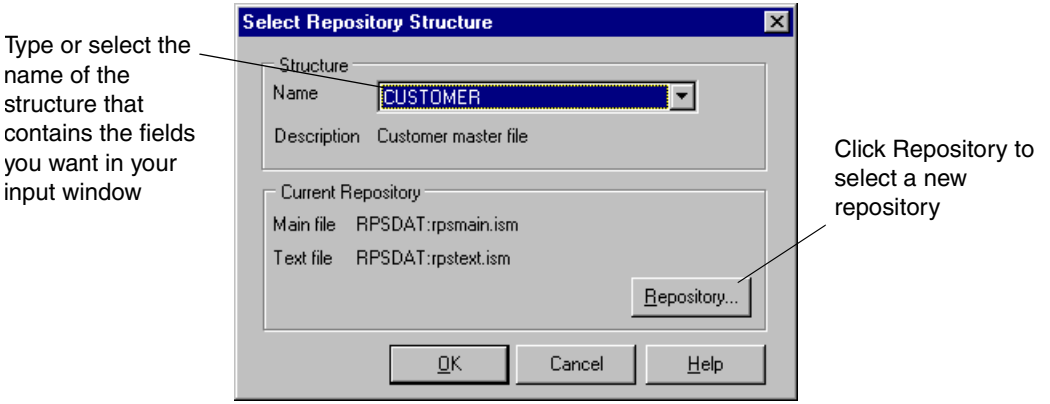



*Figure 4-6. Selecting a Repository structure.*

4. In the Name field, select the name of the structure that contains the fields you want to add to your input window.
5. Click OK to return to the Repository Fields dialog box. (See figure 4-7.)

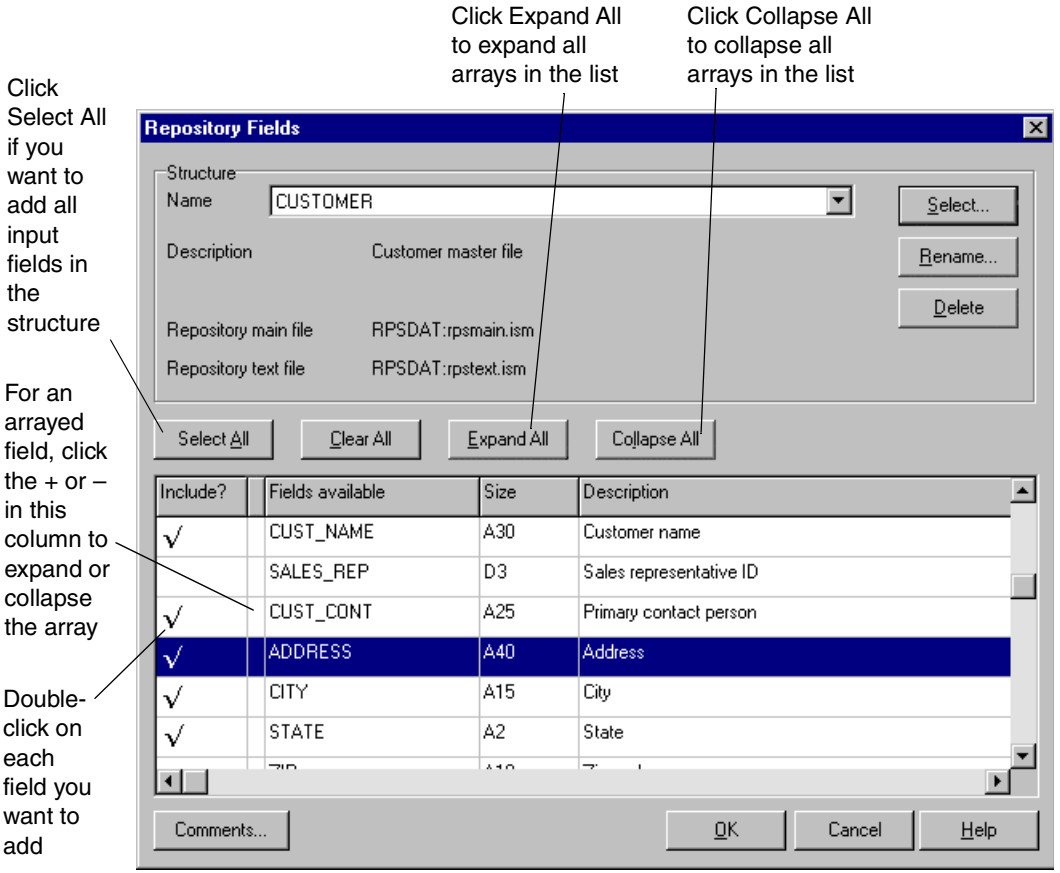



*Figure 4-7. Adding Repository fields.*

6. From the list of input fields, either double-click on each field you want to add or click the Select All button to add all input fields in the list.

   Arrayed fields are indicated by a + or – in the untitled column. Click on the + or – to expand or collapse an array. When an array is expanded, you can select individual elements of the arrayed field to add to the input window.
7. When you're finished adding fields, exit the dialog box.

## Fine-tuning your input fields

### Moving fields

Now that you see how the fields are laid out, you might want to change their positions. To move one or more fields,

1. Select the field or group of fields you want to move in the Application window. (To select a group of fields, click on the first field and then CTRL+click on all additional fields you want to include in the group.)
2. Click and drag anywhere on the field or on any field in the group (except on a sizing handle) until the field's outline is in the location you want.

### Changing a prompt

To change the prompt for an input field,

1. Select the input field in the Application window.
2. Click the Prompt property in the Properties window and type the desired text of your prompt.

## Creating a line

To add a line to your input window,

1. Select the input window by clicking on it in the Application window.
2. Click the Line button on the toolbar.
3. Click and drag on the desired location in the Application window until the line is the desired orientation and size. (Drag vertically if you want a vertical line or horizontally if you want a horizontal line.)

## Creating a command button

To add an OK button to your input window,

1. Select the input window by clicking on it in the Application window.
2. Double-click the Button button on the toolbar.

3. In the Properties window, **Text** is already displayed as the Face property, so click the Text subattribute and enter the text OK.
4. Click the Name attribute and enter the menu entry you want this button to generate, in this case I_OK.

You can repeat these steps to create Cancel and Help buttons or any additional buttons you desire. The Name properties for standard Cancel and Help buttons should be O_ABANDON and O_HELP. For nonstandard buttons, you can assign functionality in one of two ways:

- In the Name property, enter the name of a menu entry (as instructed in step 4 above).
- In the Method property, enter the name of a method routine that will perform special processing, or click the drilldown button to display the Choose Method File dialog box in Workbench so you can select or create a source file that contains or will contain the method routine code. (See "Using code templates" on page 2-26 for more information.) You can also enter the name of an executable subroutine library in the Library subproperty.

If you want to create a button that displays a bitmap graphic instead of text, simply select **Bitmap** as the Face property and then enter the name of the bitmap file, including the full directory path or logical.

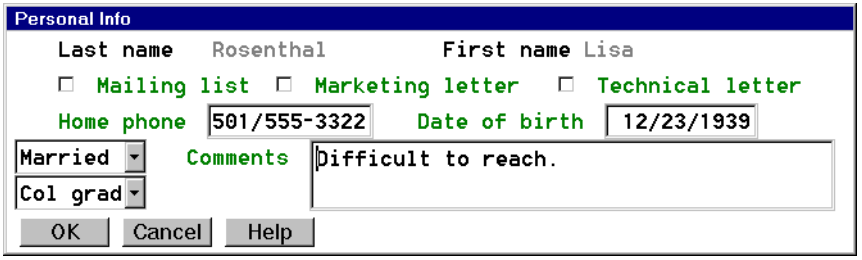
A button can also have a quick-select character, which enables the user to simulate clicking the button in your application by pressing ALT+*character*. If you want the button to have a quick-select character, click the Select character attribute and enter the quick-select character. If the button contains text, the first occurrence of the quick-select character will be underlined.

## Designing an input window from scratch

Now we're going to create an input window whose fields have not been predefined in Repository.

### Creating a "drilldown" input window

The Last field has a drilldown button, so we're going to create a second input window that's a drilldown from the first. In our application, our drilldown method for the field will display and process this second input window. Here's what it looks like:



To create the second input window in a default location,

1. Double-click the Input window button on the toolbar.

2. Click the Title property in the Properties window and enter the name of the title you desire (Personal Info, in this example).

## Creating new input fields

We haven't defined any input fields for this window in the Repository, so this time you'll need to create the fields from scratch. To add several new input fields to your input window,

1. Select the input window by clicking on it in the Application window.

2. SHIFT+click the Input field button on the toolbar.

3. For each field you want to create, click and drag on the desired location in the Application window until the field is the desired size.

4. Repeat step 3 until you've created as many input fields as you like.

5. Click any other Object toolbar button to cancel creation mode.

## Setting input field properties

### Specifying a prompt

1. Select the input field in the Application window.

2. Click the Prompt attribute in the Properties window and type the desired text of your prompt.

   If you want your prompt to be a hyperlink that calls a subroutine to perform special processing whenever the prompt is clicked, double-click the +Methods attribute to display the Hyperlink method attribute. Then click the Hyperlink method attribute and type the name of a hyperlink method subroutine you've written or plan to write, or click the drilldown button to display the Choose Method File dialog box in Workbench so you can select or create a source file that contains or will contain the method routine code. (See "Using code templates" on page 2-26 for more information.) Hyperlink prompts are displayed in green.

### Setting size

You'll probably want to make sure your input fields are the right size. To size a field,

1. Select the field by clicking on it in the Application window.

2. In the Properties window, look at the Size property.

3. If the size is not what you want, type a new size or click and drag on one of the field's sizing handles in the Application window.

### Adding a check box

You can either display an existing input field as a check box or create one from scratch using the Check box toolbar button.

To display an existing input field as a check box,

1. Select the input field by clicking on it in the Application window.
2. In the Properties window, under the Type property, set the View as subproperty to Check box.

### Adding a selection list field

Two of the fields in our example are selection list fields (sometimes referred to as "combo boxes"). You can either display an existing input field as a selection list field or create one from scratch using the Selection list toolbar button.

To create a selection list field using the Selection list button,

1. Select the input window in the Application window.
2. Double-click the Selection list button.
3. Click the Selections attribute in the Properties window and then click the drilldown button.
4. In the Selections dialog box, select **Selection list** in both the Type and View as fields. (If you have a predefined selection window, you can select **Selection window** as the Type.)
5. Replace Entry 1, Entry 2, and Entry 3 with the selection items of your choice, pressing ENTER after each.
6. Exit the dialog box.

### Adding an editable text field

The Comments field in our example is an editable text field, which is a real alpha array. When the user enters input into a text field, the text wraps automatically.

To create an editable text field,

1. Create an input field or select an existing input field in the Application window.
2. Set the Dimension subproperty (under Type in the Properties window) to a value greater than 1.

### Specifying information line text

You can display instructions or other text on the information line of your application screen when the user moves the cursor to a field. For instance, in our Personal Info example, you might want to display the text "Select the marital status of this contact" for the first selection list field.

To specify information line text for a field,

1. Select the input field in the Application window.
2. In the Properties window, set the Information line property to the text you want to display when this field is active.

### Specifying methods

You can specify routines to be called in the following situations:

| To specify a routine to be called | Use this method |
|---|---|
| Before an input field is processed | Arrive |
| After an input field is processed | Leave |
| When the field contents are being edited | Edit format |
| When the field's contents have been validated | Change |
| Before the field's contents are displayed | Display |
| When the user clicks on the field's prompt | Hyperlink |
| When the user clicks the field's drilldown button | Drill |

These method routines enable you to perform additional processing, such as validating or formatting the input. If you don't specify these subroutines, no special action is taken.

To specify method routines,

1. Double-click the +Methods attribute to display the method attributes.
2. Click the desired method attribute, and either type the name of a method routine you've written or plan to write or click the drilldown button.
3. If you click the drilldown button, Workbench will be launched (or brought to the top), and the Choose Method File dialog box will be displayed if the specified method is not in any of the source files in the current project. In the Choose Method File dialog, select or create a source file that contains or will contain the code for the method you are specifying.

   (In chapter 6, "Implementing Your User Interface with UI Toolkit," you will write sample arrive and leave method subroutines.)

# Saving your work

When you're finished defining the input windows in our exercise, you have several options:

- Close your project.
- Save or close the current script file.
- Continue defining user interface objects in this script file or create and modify other script files for additional practice.

## Closing your project

When you close your project, Composer will prompt you to save any unsaved script files, as well as the project itself. To close the active project,

1. From the File menu, select Close Project.
2. When prompted as to whether you want to save changes to your script, click Yes.

   Because your script still has a temporary name, you are prompted to enter a permanent name.
3. In the Save <Script*n*.wsc> As? dialog box, select the desired drive and directory, and type the desired script filename in the File name field.
4. Click OK.

   Composer generates a script file with the name you specified. You are also prompted to save changes to your project.
5. In the Save <Project.psc> As? dialog box, select the desired drive or directory and type the desired project filename in the File name field.
6. Exit the dialog box.

## Saving a script file

To save a new or existing script file,

- Click the Save Script button on the toolbar.

If the script has the temporary name <Script*n*.wsc> (where *n* is a number), you are prompted to enter a script filename.

### Closing a script file

When a script file is closed, all of its open objects are removed from the Application window. To close your script file,

- From the File menu, select Close Script.

If the script has not been saved since the last time you made changes, you are asked whether you want to save changes to your script.

## Compiling your script

You must compile your script before you can use it in your application. To specify compilation options,

1. From the File menu, select Compile Scripts Setup.

   The Compile Scripts Setup dialog box is displayed. (See figure 4-8.)

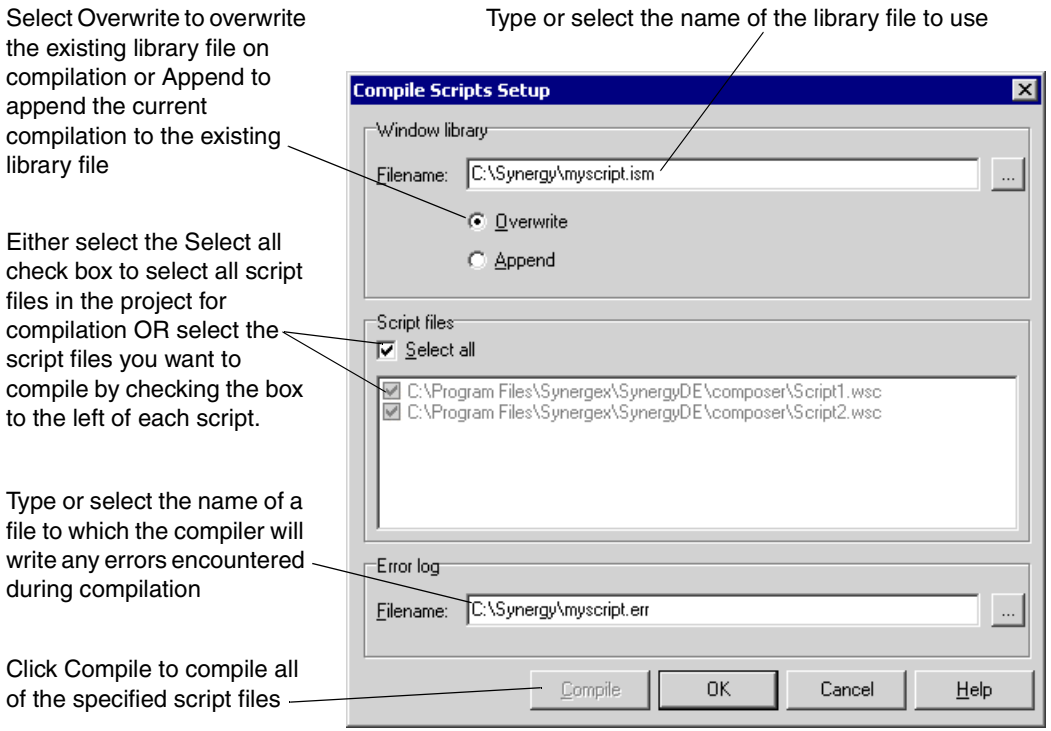

Select Overwrite to overwrite the existing library file on compilation or Append to append the current compilation to the existing library file
Type or select the name of the library file to use
Either select the Select all check box to select all script files in the project for compilation OR select the script files you want to compile by checking the box to the left of each script.
Type or select the name of a file to which the compiler will write any errors encountered during compilation
Click Compile to compile all of the specified script files
Compile Scripts Setup
Window library
Filename:
C:\Synergy\myscript.ism
Overwrite
Append
Script files
Select all
C:\Program Files\Synergex\SynergyDE\composer\Script1.wsc
C:\Program Files\Synergex\SynergyDE\composer\Script2.wsc
Error log
Filename:
C:\Synergy\myscript.err
Compile
OK
Cancel
Help

*Figure 4-8. Compiling a script.*

2. Select the scripts you want to compile and enter the desired information in the other input fields.

3. If you want to compile the selected scripts right now, click Compile. If you want to compile later, click OK to exit the dialog box.

   When you compile, the Script compiler reads the selected script files and converts them to a window library (ISAM file).

   To compile scripts later,

   - From the File menu, select Compile Scripts.

   If you have not set up any script compilation options, this command compiles the default set of scripts (which is all scripts in the project).

## Exiting Composer

To exit Composer,

- Do one of the following:
  - Click the Control Bar's Close button.
  - From the File menu, select Exit.

# 5

# Programming in Synergy Language

This chapter introduces Synergy Language and steps you through building and running a simple application. It also acquaints you with features of the Language that will help you optimize performance and simplify maintenance.


### What Is Synergy Language? 5-2

Defines what Synergy Language is and briefly describes the structure of a Synergy Language program.

### Compiling, Linking, and Running Your Program 5-5

Describes the procedures required to transform your source modules from text files into object files and finally into an organized structure that is executed by the Synergy runtime. This section also provides a short explanation of object libraries and executable libraries.

### Debugging Your Program 5-9

Explains how to include and use the debugger in your application.

### Advanced Features 5-12

Describes how to use dynamic memory and how to dispatch routines dynamically.

### Programming Tips 5-18

Provides a variety of suggestions and introduces several capabilities of Synergy Language that will help your programs run more efficiently and make them easier to maintain.

# What Is Synergy Language?

The fundamental component of Synergy/DE is Synergy Language. Most of the other elements are intended to work as layered tools on this basic component. Synergy Language provides the framework for integrating all other required components of Synergy/DE into a seamless development environment that will ultimately help you develop your total business solution. Its comprehensive capabilities and features make it easy to develop, maintain, and extend your applications.

One of the most significant benefits of Synergy Language is the fact that applications written in the Language are source-code portable to a multitude of hardware and software platforms, including Windows, OpenVMS, and popular flavors of UNIX. (Contact your Synergy/DE account manager for a full list of currently supported platforms.) In many cases, recompiling is not even necessary; in others, compiling and linking are the only requirement.

Synergy Language provides many system-level functions directly from within the language. This means you don't have to worry about getting system-level expertise to do things like set system environment variables; perform sorts; obtain system values; and create, delete, or rename files. Each of these functions is performed from within the program through a call to a system-supplied external subroutine with a standard interface that does not change between platforms. By using these routines, you can standardize your installation procedures and simplify your maintenance overhead.

In addition to providing you with a mechanism for having a single set of sources for multiple platforms, Synergy Language provides you with an excellent methodology for making data portable across platforms. If you use the Synergy DBMS utilities and take advantage of the system-supplied subroutines for ISAM file creation, creating multikeyed access files on any supported platform is quite painless. You do not need expertise in creating files on the target platform; simply run your creation program and populate your database files with the supplied utilities. The process is the same on any supported platform. Synergy/DE makes porting an opportunity, not a problem.

## Creating a source file

If you are creating your source files for the first time, it is important that you do it correctly, as you may be able to use some of these routines as templates for future development.

### Selecting an editor

Professional Series Workbench includes a Synergy Language–sensitive visual editor that automates code formatting and provides keyword recognition and completion, popup argument lists for Synergy routines, and integration with the Synergy compiler. If you're developing on Windows, we recommend you use this visual editor to simplify your editing process.

If you do not use the Workbench visual editor, you may use any popular word processing or text editing software (such as Microsoft Word or Notepad) or an editor from the command line. It really does not matter, as long as the editor you use is capable of writing out the final result as an ASCII text file, with no additional formatting or header information embedded in the file. Most word processing products will generate this type of file if you save your document as text only.

If your editing tool appends its own extension to your files, you may want to change that at the system level to take advantage of the defaults of the Synergy/DE components or to adhere to standards in your development organization. We recommend using the default extension **.dbl** for Synergy Language source files.

### Creating your modules

Once you have chosen your editing tool, you can begin creating or modifying your source code. Establishing a standard for your development is an important part of the decision process for a successful development effort. Once development has been ongoing for some time, it is difficult to retrofit standards into existing code. Choose some standard methodology and try to stick to it. You can obtain our suggested coding standard by contacting your Synergy/DE account manager.

A major benefit of Synergy Language is that you can group multiple routines in a single source module. This enables you to combine subroutines that are unique to a program within the same source module, thus reducing the number of compilations required to generate program objects. All of your definitions are maintained across the routines, requiring you to put definitions in only one place.

## Structure of a Synergy Language program

A Synergy Language program contains one or more prototype imports, structures, namespaces, classes, and routines. A Synergy Language routine is composed of two main parts: a data division and a procedure division. The data division contains most of the declarations, directives, and statements necessary to identify the various data structures that will be referenced by the program statements. The procedure division contains all of the operational statements that act on the data structures contained in the data division. Any data structure that is referenced in the procedure division must either be defined in the data division of the routine with which it is associated or in the procedure division as a stack local variable. You can define data structures that are not referenced in the procedure division, but you cannot have references in the procedure division that are not previously defined in the routine.

### Identifying your routine

Your routine should start with a statement that identifies what type of routine you are creating. The statements that identify routine types are MAIN, SUBROUTINE, FUNCTION, and METHOD. Each one of these statements tells the compiler that this is the beginning of the data division or parameter section for this routine. For a more comprehensive discussion of these statements, see the "Synergy Language Statements" chapter of your *Synergy Language Reference Manual*.

### Data division

A data division contains classes, structures, records, fields, groups, literals, temporary definitions, references to data segments located outside of the program, and possibly definitions of external functions that are referenced in the procedure division. In addition, we highly recommend that each routine contain comments that describe the purpose of the routine and define any interactions with other routines. If the routine is an external subroutine, function, or method, the arguments that are passed to and returned from the routine should also be defined.

The end of the data division and the start of the procedure division is defined by the PROC statement.

### Procedure division

The procedure division contains the logic and operational statements that will be executed to make your program perform the task you desire. Each statement will be executed according to the syntax and semantics defined in the *Synergy Language Reference Manual*. Synergy Language allows for data manipulation, file and terminal I/O, program logic control, arithmetic operations, and conditional execution. By combining these capabilities, you should be able to create a routine (or group of interacting routines) that solves any business problem.

The end of the procedure division is defined by the END, ENDMAIN, ENDSUBROUTINE, ENDFUNCTION, or ENDMETHOD statement. These statements tell the compiler that the routine is complete and prepares it to deal with the next routine.

## See also

Your *Synergy Language Reference Manual*, which contains the syntax and specifications for every statement and system-supplied subroutine in Synergy Language. Refer to this manual for further information.

# Compiling, Linking, and Running Your Program

Once you've written your source modules, you must compile, link, and run your programs. Here are the steps you should follow:

1. Compile each source module. See "Compiling your program" below.
2. If desired, combine your compiled routines in an object library. See "Creating object libraries" on page 5-6. Creating an object library simplifies the linking process (as well as your distribution strategy).
3. Link your compiled routines or object library, generating an executable library if desired. See "Linking your program" on page 5-6.
4. Test your program by running it. See "Running your program" on page 5-8.

## Compiling your program

The Synergy compiler interprets the ASCII code you have written in your source modules and, using the rules of Synergy Language, creates another file that contains the instructions to perform the program actions you have specified. The output from the compiler is called object code and is sometimes referred to as an object file. Most applications are composed of many of these individual object files that are bound (or linked) together into a single executable program.

By default, the compiler expects the input source module to have the extension **.dbl**. If your programs follow this convention, you will not have to specify a filename extension in your compiler commands. The compiler creates output files with the following default extensions:

| Extension | File type |
|---|---|
| **.dbo** | Generated object file |
| **.lis** | Listing file |

We recommend using these default filename extensions so that other tools (for example, Windows Explorer) will know what type of file it is.

For instructions on compiling your program, refer to the "Building and Running Synergy Language Programs" chapter in *Synergy Language Tools*. You can also compile your program directly from Professional Series Workbench by doing the following:

1. Make sure the file you want to compile is open.
2. From the Project menu, select Compile.

There are many other ways to request a source compile on various platforms. For example, Serena ChangeMan Builder's **make** command is an excellent choice for program builds.

## Creating object libraries

Once you've created a number of related external subroutines and have tested them thoroughly, you may want to group them together so you can reference them as a single entity. Instead of having to remember to include all of the object files that are referenced in each program every time you link the application, you can create an object library using the Synergy librarian.

An object library is a collection of object files that have been compiled and stored in a special file that can be referenced by the Synergy linker. Any routines that are included in the object library will be extracted from the library at link time. Thus, instead of distributing hundreds of **.dbo** files, you can distribute a single **.olb** file that contains all of the compiled source code routines.

For instructions on creating an object library, refer to the "Building and Running Synergy Language Programs" chapter in *Synergy Language Tools*.

## Linking your program

After you've compiled your source code into object files, you must gather the object files into a cohesive unit so they can be executed as a single program. This gathering is done by the linker program. The linker also performs many other functions, such as resolving addresses for global symbols, figuring out where external data references are located, identifying the exact location of external routines so your program can call them, and associating all program information with the required system resources for your system. The output from the linker is an executable program: a complete, self-contained group of instructions capable of being executed by the runtime.

By default, the Synergy linker expects your object files to have the extension **.dbo**. If your programs follow this convention you will not have to specify a filename extension in your linker commands. The linker creates output files with the following extensions:

| Extension | File type |
|---|---|
| **.dbr** | Executable file |
| **.map** | Map file |
| **.elb** | Executable library |

If any of your object files have been changed, you must recompile them before relinking your application for the changes to take effect. You do not need to recompile object files that have not been changed. When you relink your application, your changes will be incorporated into the new executable program.

For instructions on linking your program, refer to the "Building and Running Synergy Language Programs" chapter in *Synergy Language Tools*. You can also link your program directly from Professional Series Workbench by doing the following:

1. Make sure the file you want to link is open.
2. From the Project menu, select Build.

   If the file was not previously compiled, selecting Build will compile it.

## Creating executable libraries

As useful as object libraries are, they have limitations. If your source code changes, not only do you have to recompile the module, but you also have to relink every application that included that particular object file. This could be time-consuming if your application is large and the object file you have changed is common to most of your programs.

To address this issue, as well as others, Synergy Language provides a feature called executable libraries. An executable library is different from an object library in that the object file is not included in the executable program. Instead, the linker includes a referencing pointer to where the routine is located in the executable library. When the Synergy runtime executes your program and the routine is called, the runtime knows how to execute the code from the executable library.

Four advantages of using an executable library instead of (or in addition to) an object library are

- smaller programs. If you have a subroutine that is called by several programs and you're not using an executable library, each executable program contains a complete copy of the compiled subroutine. As a result, your programs are larger in size and use more memory when executing. Using an executable library can significantly reduce the size of your program images.
- time savings. As with an object library, if you change an object file, you have to recompile the file and replace the old object file with the new one. However, you don't have to do anything else. Because your programs contain only a pointer to the object file to be executed, and the executable library keeps track of where the new object is located, you don't have to relink your programs to take advantage of the change. When your program is executed, the Synergy runtime merely gets the new copy from the executable library.
- more efficient distribution process. Because your executable program contains only a referential pointer to another program image, you can simply replace the executable library, instead of redistributing an object library and the commands necessary to relink the application. When the new executable library is installed, your changes are already in effect.
- creation by the Synergy linker. You can either use an existing object library and create an executable library that contains all of your object files, or you can have the linker put the object files directly into an executable library.

For instructions on creating an executable library, refer to the "Building and Running Synergy Language Programs" chapter of *Synergy Language Tools*.

### Restrictions

Once an image is inserted into an executable library, it cannot be removed. If you want to delete some images from an executable library, you must delete the library and rebuild it from scratch.

To facilitate the process of rebuilding an executable library, you might want to keep an object library from which you can regenerate your executable library. The Synergy librarian provides the necessary tools for managing individual object files in an object library.

### VMS

Executable libraries are not implemented on OpenVMS systems. On those systems, the functionality has been implemented as OpenVMS shared images.

## Running your program

Now you've come to the easiest part: actually running your program. This is the final step in program development. If you've done your job completely, you will have a working program. If not, you will have to figure out where changes need to be made, change the source code, and then recompile, relink, and retest your program.

By default, the Synergy runtime expects your programs to have the extension **.dbr**. If your programs follow this convention, you will not have to specify a filename extension in your runtime commands.

For instructions on running your program, refer to the "Building and Running Synergy Language Programs" chapter in *Synergy Language Tools*. You can also run your program directly from Professional Series Workbench by doing the following:

1. Make sure the file you want to run is open.
2. From the Project menu, select Execute.

# Debugging Your Program

No one writes perfect code, at least not all of the time. Because almost every program has a few bugs that need to be fixed, a debugger is a valuable tool.

Built into Synergy Language is a powerful utility that enables you to interactively control program execution, examine the data structures referenced by your program, modify those data structures to contain whatever values you choose, and validate memory references for correctness. You can instruct the debugger to "watch" a variable and report when its value changes, what the old value was, and what the new value is. You can identify a particular statement (or multiple statements) as a stopping place. You can decide whether to step through routines or execute them in their entirety.

To run a program under the control of the Synergy debugger,

1. Compile your program with the **-d** option.
2. Link your program with the **-d** option.
3. Run your program with the **-d** option.

You can also run a program in debug mode directly from Professional Series Workbench by selecting Debug from the Project menu.

Much of debugging is common sense: finding out where the program is going wrong and stopping the execution before the problem happens, then stepping through the program one statement at a time, looking at variables that affect the processing, changing values to create conditions that are potential problems, and trying to validate your program logic by tracing the program flow. As you interact with your programs, you'll get a sense of what kinds of mistakes cause what classes of problems, and identifying the source of problems will become easier.

## See also

The "Debugging Your Synergy Programs" chapter of *Synergy Language Tools*. We highly recommend reading this chapter to become familiar with the different commands and capabilities of the debugger, and for instructions on running the debugger remotely. The debugger is a tool that will save you a lot of time if you use it correctly.

## Saving and restoring debugger settings

You can save the current break points, watch points, and option settings of the debugger so these settings can easily be restored the next time you run the debugger.

Use the SAVE *file* debugger command to save debugger settings to the specified file. Use the @ *file* debugger command to restore the saved settings. See the "Debugging Your Synergy Programs" chapter of *Synergy Language Tools* for details about these commands.

For information about saving and restoring the current debugger state to a file from Professional Series Workbench, see "Debugging a project" on page 2-31.

# Debugging with bounds checking

Bounds checking is a very powerful feature of the debugging capabilities in Synergy Language. Running your application with bounds checking enabled will identify any situations in which data is being stored into structures that are referenced beyond the "bounds" of their normal definition. Trying to monitor situations like this manually is a debugging nightmare.

Many of the legacy languages (ANSI DIBOL and Synergy DBL) take advantage of the capability that allows you to reference subsequent data structures as a subscripted array element of a previous data structure. While this may have been a solution to some problems, it opened the door to a much bigger problem: unknowingly writing or referencing data beyond expected limits. In some cases these data reference operations are detected by the system when the program is run, resulting in errors. In other cases the reference operation goes undetected and remains in the application like a time bomb, waiting for the most inopportune time to overwrite a critical variable that will cause another routine that depends on that value to malfunction. Debugging one of these "transient" error conditions is one of the hardest and most frustrating exercises in software engineering.

Synergy Language will find this class of error for you automatically if you compile your application with bounds checking enabled. When the program is run, it will report any and all references that result from an "out-of-bounds" condition.

When you compile your programs with bounds checking enabled, all pseudo arrays (such as a field defined as **10d5**) are converted to real arrays (for example, **[10]d5**). While you are in development mode, you should build all routines in your application with bounds checking enabled. When you are ready to release your application to production, simply compile without bounds checking enabled, and the size of the image will be reduced significantly. Below are examples of conditions that will be reported when a program is compiled with bounds checking enabled.

## Examples

When the following program is run, the runtime will report an out-of-bounds condition, because the code is actually referencing **var[1](1,10)**, the first element of the array, which has a "real" length of 2. Because you are trying to write 10 characters into a two-character field, the runtime will report an error.

```
main myroutine
record
    var       ,[10]d2

proc
    var(1,10) = "0001020304"
endmain
```

The following example will report an out-of-bounds condition if the subroutine is compiled with bounds checking enabled and the calling routine is compiled with bounds checking disabled. This example illustrates why it's important to compile all your routines with bounds checking enabled.

```
record
    var   ,10d2
    .
    .
    .
    xcall subr(var)
    .
    .
    .
subroutine subr
    arg   ,d
    .
    .
    .
    arg(2)  = 10
```

The first statement below will generate an out-of-bounds condition, because **ivar** is defined as being **i4**. The correct way to clear both of these integer variables at the same time is to reference the named record as an integer, as the second statement does.

```
record name
    ivar   ,i4
    ivar2  ,i4
    .
    .
    .
    clear ivar(1,8)
    clear ^i(name)
```

Both of the statements below will generate an out-of-bounds condition. While both types of data reference are common techniques in many legacy programs, they are references outside the bounds of the definition of **var** and, as such, will be reported.

```
record
    var       ,d2
    var2      ,d2
    .
    .
    .
    var(1,4) = 1010
    var(2) = 10
```

# Advanced Features

## Using dynamic memory

Most applications need to allocate space to store varying amounts of data. In the past, this was accomplished by considering what the "worst case" would be and defining sufficient storage space to accommodate this situation. This is quite inefficient for two reasons. The first is that it causes the program to require more memory at runtime and more storage space on the disk. The second is that if the application at some point in time exceeds the original estimate of the "worst case," the code has to be changed, with all of the inherent inconvenience associated with code change.

A better solution is to allocate memory dynamically as it is required by the program. When the program requires memory, it simply tells the system how much it needs. The system allocates the memory and returns a pointer to the base of the memory segment. We refer to this pointer as a *handle*. All future references to this memory will use this handle. There is no practical restriction on the amount of memory available, and once used, the memory can be released back to the system. The amount of memory is totally dynamic: it can grow or shrink as required by the application. As new memory is allocated, it is logically appended to the existing memory, and as memory is released, it is logically truncated from the existing memory.

### Defining data handles

The pointer for each memory segment must be allocated in the data division. Multiple segments are allowed, but each segment must have its own handle. A handle is an **i4** integer and should be aligned on an even longword boundary for maximum performance:

```
.align long
record
    handle1  ,i4          ;The handle for one segment
    handle2  ,i4          ;The handle for another segment
```

### Defining memory structures

Memory allocation and reference is done in conjunction with structures. Structures are defined in the data division using the STRUCTURE statement and function the same as the Synergy Language RECORD statement, except they do not allocate any space in the program. Structures can be .INCLUDEd from the repository, .INCLUDEd from files, or hard-coded in your application.

```
structure cards
  value      ,i4
  suit       ,a1
  rank       ,a1
```

## Manipulating dynamic memory

Dynamic memory is manipulated by the %MEM_PROC function, which performs four operations:

- Allocation
- Resizing
- Deallocation
- Registration

### Allocating memory

Memory allocation is performed by specifying the DM_ALLOC value as the first argument to the %MEM_PROC function and any decimal expression as the second argument. The function value is assigned to the **i4** variable defined as the handle for the memory segment. The second argument is the number of bytes of memory that will be allocated. For example:

```
handle1 = %mem_proc(DM_ALLOC, ^size(cards)*52)
```

Using the structure definition from the previous example, this code will allocate 312 bytes of memory (6*52) and return the pointer to the memory in the integer named **handle1**. The contents of the memory segment are undefined.

### Resizing memory

Existing memory segments can be expanded or collapsed by specifying the DM_RESIZ value as the first argument to the %MEM_PROC function, a decimal expression representing the new size of the memory segment as the second argument (either greater than the original or smaller than the original), and the handle of the original memory segment as the third argument. The return value will be the new handle of the memory segment. Here's an example of expanding memory:

```
handle1 = %mem_proc(DM_RESIZ, ^size(cards)*52*4, handle1)
```

Using the previous definitions, this code will allocate 1248 bytes of memory and take the contents of the original 312-byte memory segment and copy it to the first 312 bytes of the new memory segment. The contents of the remaining memory in the segment is undefined. The pointer to the new memory segment will be returned in the variable **handle1**.

To return unwanted memory to the system, simply specify a lesser value as the second argument. For example:

```
handle1 = %mem_proc(DM_RESIZ, ^size(cards)*52*2, handle1)
```

This code will remove the last 624 bytes from the memory segment. The size of the memory segment will now be 624 bytes. Any information that existed in the returned memory will be lost. The pointer to the memory segment will be returned in the variable **handle1**. Do not assume that this pointer will be unchanged.

### Deallocating memory

To return memory to the system, specify the DM_FREE value as the first argument to the %MEM_PROC function and the handle of the memory segment as the second argument. The function value will be returned as zero (0), and we recommend that this value be assigned to the handle. Any data reference operations against a handle with a zero value will generate an error condition that can be trapped. For example:

```
handle1 = %mem_proc(DM_FREE, handle1)
```

This code releases the memory that was previously allocated and sets the value of **handle1** to zero.

### Registering memory

Your program may call external routines written in other languages or system functions that return pointers to memory segments that were allocated outside of Synergy Language. You can reference these segments from within your program if they are registered within your program. The process of registration does not allocate memory; it makes memory segments that are allocated externally available to your program as if they had been allocated by your program.

To register memory, specify the DM_REG value as the first argument to %MEM_PROC, a decimal expression that represents the size of the memory segment as the second argument, and a platform-specific-sized integer address of the memory segment as the third argument. The value of the function will be the handle of the memory segment. For example:

```
.align quad
record
    addr      ,D_ADDR
    size      ,i4
    .
    .
    .
proc
    .
    .
    .
    xcall some_routine(%val(size),%ref(addr))   ;Call an external routine
    handle1 = %mem_proc(DM_REG, size, addr)     ;Register the memory
```

This code will register the address that was passed into the variable **addr**. Expressing the type and size of this variable by referencing the .DEFINEd value D_ADDR ensures that this variable will match the system requirement of all platforms. On 64-bit systems it will allocate an **i8** field; on 32-bit systems it will generate an **i4** field. D_ADDR is conditionally defined for you by the compiler. After the %MEM_PROC function call, the memory segment will behave exactly as if it were allocated from within this program.

## Accessing memory

Once memory has been allocated, we use the ^M data reference operation to address the memory locations. In effect, a memory segment can be considered to be a dynamic array that can be addressed in the same manner as a multidimensioned group.

The following example uses the %MEM_PROC dynamic memory function to allocate a dynamically sized array, which will be resized as needed. The routine reads the records from a customer file into memory and then sorts the records in memory and prints them out in sorted order. This example presumes that the developer can supply the local routine "print" and the QSORT sort routine "sort_by_past_due." (Refer to QSORT in the "System-Supplied Subroutines and Functions" chapter of the *Synergy Language Reference Manual* for more information.) Note that this routine does no error trapping, so it presumes that the customer records will all fit into available memory.

```
.align
.define D_INITIAL_ALLOC    ,10    ;Initially allocate for 10 records max
.define D_INCR_ALLOC       ,5     ;If we need to resize, go up by 5
.define D_CUST_CHAN        ,12    ;Customer channel number
.define D_FMT_MONEY        ,"Z,ZZZ,ZZZ,ZZX.XX-"

record
    handle          ,i4          ;Handle to dynamic memory
    nmalloc         ,i4          ;Number of records allocated
    nmloaded        ,i4          ;Number of records actually loaded
    ix              ,i4          ;A scratch index

structure customer               ;Customer record structure
    group rec       ,a
      number        ,d6
      name          ,a30
      address       ,[3]a30
      current_due   ,d12.2
      over_30       ,d12.2
      over_60       ,d12.2
      over_90       ,d12.2
    endgroup

record pline                     ;Print line buffer
    pcust           ,a6
                    ,a2
    pname           ,a30
                    ,a2
    pover90         ,a17
                    ,a2
    pover60         ,a17
                    ,a2
    pover30         ,a17
```

```
proc

;First, allocate an array and read in the customer records

    nmalloc = D_INITIAL_ALLOC           ;Allocate for 10 records
    handle = %mem_proc(DM_ALLOC, nmalloc)
    clear nmloaded                      ;None loaded yet

    open(D_CUST_CHAN, "I:I", "DAT:customers")  ;Open customer file
    repeat
      begin
        if (nmloaded .ge. nmalloc)      ;Need to increase array size?
          begin
            nmalloc += D_INCR_ALLOC     ;Bump it by 5
            handle = %mem_proc(DM_RESIZ, nmalloc, handle)
          end
        incr nmloaded;Loading one now
        reads(D_CUST_CHAN, ^m(customer[nmloaded].rec, handle), eof)
      end
eof,
    close D_CUST_CHAN                   ;Close customer file
    decr nmloaded                       ;EOF on the last one

;Now do an in-memory sort based on some complex criteria

    xcall qsort(^m(customer.rec, handle), nmloaded, "sort_by_past_due")

;Now print them out in that order

    for ix from 1 thru nmloaded
      begin
        clear pline
        pcust = ^m(customer[ix].number, handle), "XXXXXX"
        pname = ^m(customer[ix].name, handle)
        pover90 = ^m(customer[ix].over_90, handle), D_FMT_MONEY
        pover60 = ^m(customer[ix].over_60, handle), D_FMT_MONEY
        pover30 = ^m(customer[ix].over_30, handle), D_FMT_MONEY
        call print
      end

    xreturn  ;Because the allocation was not static, this releases it
end
```

### Static vs. volatile handles

Handles are classified as one of two types, static or volatile:

- Static handles are persistent and do not get released when the activation level of the generating routine is exited. They are stored as positive numbers.
- Volatile handles are dynamic and go away along with the routine activation level. They are stored as negative numbers.

Both static handles and volatile handles go away when the entire program is exited or the handle is explicitly released.

## Dispatching routines dynamically

Synergy Language provides a system routine (XSUBR) that enables programs to determine at runtime what external routine will be called. This capability provides a multitude of opportunities for the application developer to provide flexibility in routines. Consider the case where the name of a routine to be called is returned to a UI Toolkit program with a special prefix. You could add a completely new feature to the code simply by changing the menu column in the window library and adding the routine to an associated executable library. You wouldn't have to change any code or even relink the application.

Consider the following script definition:

```
.column functions,  "Functions"
.entry s_rolodex,   "Rolodex
.entry s_calculate, "Calculator"
.entry s_diary,     "Diary"
.entry s_mailbox,   "Mail"
.end
```

The code to process this script could be as simple as the following:

```
xcall m_process
if (g_select)                    ;Menu selection
    case g_entnam of
      begincase
"O_EXIT":  exitloop
"O_ABOUT": xcall u_about(TITLE, VERSION,%datecompiled)
"S_":      xcall x_subr(g_entnam - "S_")
      endcase
```

Adding a new routine would not require changing this code at all. For instance, an application could have matched sets of window libraries and executable libraries, distributed as pairs, that would work with the same executable image but provide very different capabilities. If the application went one step further and built the menu columns on-the-fly, simply updating a database file and the executable library would allow calls to the new routines.

# Programming Tips

## Referencing data indirectly

Environment variables (or logicals) give your application tremendous power to reference data in an indirect manner. You should use environment variables to simplify your coding and standardize the way you address the location of the files that are opened by your application. In most cases, environment variables are a convenience, but in some cases they are a necessity. For instance, there is a 31-character restriction on the file specification for the executable library that is passed to the Synergy linker. If your full path exceeds this limitation, your link command will return an error.

In terms of convenience, environment variables enable you to reference different file locations without having to change your source code, which brings with it the associated overhead of recompiling and relinking your code. In addition to the time lost, there is always the possibility that you may have made a typographical error, executed an erroneous compile or link command, or forgotten to include a required file. Using environment variables enables you to adjust where the database files are located and then execute your program, with the same program referring to a different set of data.

### Three cases for environment variables

One of the most powerful uses of this tool is that you can code your application to point to a data set that is referenced by an external environment variable and test your application against a test data base. When you're done testing, you simply reset the environment variable; the application now points to the real data, without the need for you to touch the application in any way.

Another great feature of this paradigm is that your application may be installed at multiple customer sites. Instead of having to go into each program module that contains a file specification and customize the file references to the individual site, you can just use a set of environment variables in your application and set them at each customer site. Not only do you have a faster and more standard installation procedure, but you also have a single set of sources for multiple installations.

A third reason to use environment variables is that on different operating systems the file specifications are formatted differently. Consider the following actual file specifications that reference a Synergy ISAM file that must be opened in an application on three different platforms:

| System | File specification |
|---|---|
| Windows | **c:\myapp\finance\gen_ledg.ism** |
| UNIX | **/usr/apps/myapp/finance/gen_ledg.ism** |
| OpenVMS | **user$disk1:[myapp.finance]gen_ledg.ism;20** |

You could take advantage of environment variables by coding the OPEN statement to refer to this file on all systems in a generic manner. Your application would address this file with a statement like the following:

```
open(chan_no,"I:I","FINAPP:gen_ledg")
```

By simply setting an environment variable named FINAPP on each system, your application would open the correct file regardless of the difference in format for each system. Note also that we have taken advantage of the default file extension capabilities and have not specified an extension for the file.

## Setting environment variables

How environment variables are set is system-dependent, as each operating system requires a somewhat different syntax, but the basic concept is the same. On each system, we tell the operating system the name we are going to use as the environment variable, and we also tell it what we want the environment variable to mean when we use it in our application or command line references.

In the example above, we would set the environment variables in the following manner:

| System | Environment variable definition |
|---|---|
| Windows | `set FINAPP=c:\myapp\finance` |
| UNIX | `FINAPP=/usr/apps/myapp/finance ;export FINAPP`<br>or<br>`setenv FINAPP /usr/apps/myapp/finance` |
| OpenVMS | `define FINAPP user$disk1:[myapp.finance]` |

As you can see, depending on your system, you will need to set the environment variables in different places and using slightly different syntax. Consult the "Environment Variables" chapter of *Environment Variables and System Options* for a more comprehensive discussion on how to set environment variables on your system. Where you set your environment variables is usually a function of the scope that your environment variables must have. If they are going to be needed by everyone who is on the system, it may be useful to set them in some system initialization file. If they are going to be used by just a few individuals, it may be better to define them in individual log-in command (or **.profile**) files. Talk with your system administrator to decide where and how environment variables should be set in your particular environment.

# Comparing data

Two separate groups of relational operators are available to the Synergy Language developer to evaluate the relationship of alpha data. Most developers know of the first group but are not aware of the significant difference in the behavior of the second group. The two groups and their operators are as follows:

| Relational operators | String relational operators |
|---|---|
| .EQ. | .EQS. |
| .LT. | .LTS. |
| .GT. | .GTS. |
| .NE. | .NES. |
| .GE. | .GES. |
| .LE. | .LES. |

The difference in behavior between these groups is based on the length of the comparison that is made. Relational operators perform their comparison based on the shorter of the two operands. String relational operators perform their comparison based on the longer of the two operands, with the shorter being logically extended with blanks (spaces) until the shorter operand is the same size as the larger operand. This means that comparing “A” to “ABC” with .EQS. is logically equivalent to comparing “A  ” to “ABC”, and the resulting truth value is FALSE. Performing the same comparison with .EQ. is logically equivalent to comparing “A” to “A”, and the resulting truth value is true. This behavior holds for all of the string relational operators.

This distinction is most important when considering how the CASE and USING statements are evaluated. These statements use the relational operator .EQ. in performing their comparisons. This is a double-edged sword: your application may take advantage of this feature to generate matches for multiple values with a single entry, or your application may not behave as expected and may match on values that were not intended. Consider the following sample code:

```
case value of
  begincase
"A":   call a_process      ;Catches every entry that starts with "A"
"AB":  call ab_process     ; That means you can never get here!
"ABC": call abc_process    ; Or here!
  endcase
```

The CASE and USING statements perform evaluations in a top-down manner. Therefore, in the above code, any value that would match the second or third case would by definition have to match the first. To make this work correctly, the order of the cases must be reversed as follows:

```
case value of
  begincase                ;Matches the following values
"ABC": call abc_process    ; ABC, ABCC, ABCD, ABCDE...
"AB":  call ab_process     ; AB, ABD, ABE, ABB...
"A":   call a_process      ; any remaining values starting with A
  endcase
```

Understanding the behavior of the different relational operators will enhance your productivity and provide you with more options as you code your application.

## Manipulating dates

Synergy Language provides the capability to transform dates to and from a numeric representation, which enables you to perform mathematical operations on the dates. Adding and subtracting periods of time, determining differences between dates, and determining what day of the week a particular date falls are simple function calls. The algorithm is based on the Gregorian calendar and calculates the days since December 31, 4713 BC.

Four function calls provide this capability:

- %JPERIOD converts a date to a numeric value.
- %DATE converts a numeric value to DD-MMM-YY (or YYYY) alpha format.
- %NDATE converts a numeric value to YYYYMMDD numeric format.
- %WKDAY returns the ordinal number of a weekday (Sunday is 1).

### Converting dates to a numeric value using %JPERIOD

%JPERIOD accepts either an alpha representation of the date in the form DD-MMM-YYYY or a numeric representation of the date in the form YYYYMMDD. If a date is not passed as an argument, the system date is taken as the default.

Consider the following statements:

```
integer = %jperiod("25-DEC-1997")
integer = %jperiod(19971225)
integer = %jperiod
xcall jperiod(integer, 19970101)
```

Both the first and second statement are functionally equivalent and would store 2451125 in the integer variable. The third statement would store the numeric value of the current system date in the integer variable. The fourth statement is an example of how this function can be called as an external subroutine, specifying a variable as the first argument. The integer variable would contain 2450767.

## Converting numeric values to alpha dates

%DATE transforms any numeric value to its equivalent alpha representation of the date formatted as DD-MMM-YYYY, DD-MMM-YY, or a truncated version of the latter. If no value is passed, it will return the current system date as the default. If the length of the alpha variable is too small to contain the entire date, it will be truncated.

```
record
    avar11    ,a11
    avar9     ,a9
    avar6     ,a6
.
.
.
avar11 = %date(2450767)    ;avar11 = 01-JAN-1997
xcall date(avar9,2451125)  ;avar9 = 25-DEC-97
xcall date(avar6,2451125)  ;avar6 = 25-DEC
```

## Converting numeric values or alpha dates to numeric dates

%NDATE transforms any numeric value to its equivalent numeric representation of the date formatted as YYYYMMDD or YYMMDD. Alternatively, it converts an alpha formatted date to a numeric formatted date. If no value is passed, it returns the current system date as the default. If the length of the destination is too small to contain the entire date, the date will be truncated.

```
record
    avar4     ,a4
    avar6     ,a6
    avar9     ,a9
.
.
.
avar4 = %ndate(2451125)         ;avar = 1225
avar6 = %ndate("25-DEC-1997")   ;avar = 971225
xcall ndate(avar9,2451125)      ;avar = 19971225
```

## Converting numeric and alpha dates to a day of the week

%WKDAY takes either a numeric formatted date or an alpha formatted date and returns the ordinal day of the week for that date. If the alpha format is used, any portion of the date that is omitted will default to the system date.

```
record
    day             ,[7]a*,    "Sunday", "Monday", "Tuesday", "Wednesday",
  &                              "Thursday", "Friday", "Saturday"
    group crimbo    ,d8
      year          ,d4
      month         ,d2
      crimbo_day    ,d2
    endgroup
```

```
proc
    open(1, o, 'tt:')
    crimbo = %ndate("25-DEC")
    incr year
    writes(1, "Christmas is on a " + %atrim(day[%wkday("25-DEC")]) +
  &        "; next year it is on a " + %atrim(day[%wkday(crimbo)]) + ".")
end
```

If this program were run in the year 1997 (the current year is assumed because the year is omitted), the result would be "Christmas is on a Thursday; next year it is on a Friday."

## Using compile-time definitions

Your application can take advantage of compile-time definitions to provide a level of indirection in your code as well as make the code much more self-documenting. An additional benefit is that compile-time definitions do not allocate any space in the data division. These definitions can be contained in external files that you can conditionally include into your programs or code directly into your source programs. Thus, you can edit your code by modifying the value of the definition in a single-source module that is included in your source programs.

## Using integer data

For performance reasons, we recommend that you use longword-aligned **i4** variables for any variable in your program that won't be stored on disk. This includes, but is not limited to, variables for things like loop counters, values that are tested for conditionals, and variables used in mathematical computations. Many RISC platforms do not provide native math instructions that deal with nonaligned integers, so Synergy Language uses emulated math operations if the integers are not on a longword boundary. By using aligned integers, you will realize a significant improvement in performance. We strongly recommend that you do not store integer data in your data records, as this could impede your ability to migrate your software to other platforms in the future.

## Using CASE vs. USING

Synergex recommends using the USING statement, which always generates the most efficient code. The structures that are available (matching on ranges, matching on multiple values, matching on expressions, matching the null case, matching on variables) in the USING statement have always made it more powerful and more functional; now the performance matches the functionality.

Using the CASE statement is only practical when the match strings are a series of alpha literals. A further restriction on the CASE statement is that it must be bounded by a BEGIN-END block if it is nested within an IF-THEN-ELSE structure. Even if a CASE statement is not currently within an

IF-THEN-ELSE structure, Synergex recommends encapsulating all CASE statements within a BEGIN-END block to prevent possible confusion should an IF-THEN-ELSE be introduced into the code or that code segment be moved or copied to another location.

# 6

# Implementing Your User Interface with UI Toolkit

This chapter introduces S/DE UI Toolkit and its relationship to application development with Synergy/DE. It explains features, describes programming techniques, and provides sample code that can be used as a template for your own development. S/DE Repository is used to define the data for our sample application that we began in chapter 3.


### What Is UI Toolkit? 6-3

Provides an overview of the capabilities of UI Toolkit.

### Starting UI Toolkit 6-4

Explains the UI Toolkit screen and the initialization of the Toolkit environment, as well as the style of programming necessary for application development with UI Toolkit, how to let Toolkit manage your files, and using variables and defined values.

### Managing Display Levels with Environment Processing 6-8

Defines Toolkit environments and describes techniques to save and restore program displays.

### Managing Window Libraries to Store and Retrieve Display Components 6-10

Describes the definition, creation, and use of window libraries.

### Creating Script Files and Window Libraries 6-11

Explains how to create menu columns, windows, and lists and how to write code to process them.

### Implementing Online Help 6-35

Explains how to add online help to your application.

### Organizing Your Display with Tabbed Dialogs 6-39

Describes how to create and process tab sets from within a UI Toolkit application.

### Using Composite Windows to Combine Windows and Lists 6-42

Describes how to combine multiple windows and/or lists into a composite window.

**Using Methods 6-45**

Defines methods and how to apply them to add power and flexibility to your UI Toolkit application.

# What Is UI Toolkit?

S/DE UI Toolkit is a collection of external subroutines and functions that can be called from your programs. It is not another language but rather an extension to Synergy Language that enables you to create and process screens for your applications. The Toolkit is involved in every aspect of the terminal user interface. Enhanced capabilities include data presentation, interactive data input and editing, Windows pull-down menu processing, demand-loaded dynamic lists, and tabbed dialogs. On Windows systems, the UI Toolkit manages mouse interactions, providing your application with a truly native "Windows" look and feel. (For more information about Windows-specific UI Toolkit features, refer to the "Developing for Windows" section of your *Professional Series Portability Guide*.)

Your application is still a Synergy Language program. UI Toolkit features are contained in an executable library (**WND:tklib.elb** on most systems) against which your application is linked. These Toolkit routines perform the appropriate operation on each of the platforms supported by Synergy Language. What this means to your application is that your screen I/O is platform independent. Regardless of where your application is run, your code does not have to change; it lets the Toolkit routines handle any system differences. It is even possible to create a single set of source files that runs as a Windows application on Windows systems and as a cell-based application in a non-Windows environment.

UI Toolkit does not change your program logic beyond the terminal interface. Using Toolkit does not change the way you handle the data once it has been entered by your user. So if you plan to convert an existing application, much of your existing code may not need to be replaced.

## Performing terminal I/O

To use UI Toolkit, you must change your perception about how terminal I/O is performed. All terminal I/O is done through the Toolkit routines. Instead of doing I/O at the field level, you will do it at the record level. Also gone is the need to position the cursor, take the input, refresh the screen after errors, and then process the data. Most data validation can be handled by Toolkit as well. Gone is the need to manipulate each field in a record on an individual basis. Your window definition can specify attributes for an entire record. When processing the window, you specify a data area and pass control to Toolkit. When Toolkit returns, you have a complete, correct data record, with all fields entered and verified for accuracy. All that remains for you is to process that complete record.

# Starting UI Toolkit

To use UI Toolkit, your program must call U_START, the Toolkit's start-up routine. U_START initializes the Toolkit's memory, opens the terminal, opens the window library where window definitions are contained, and establishes the parameters of the Toolkit screen. Other options set the size of the display regions, establish upper and lower bounds for Toolkit channel allocation, and set the maximum number of levels for environment processing.

## The Toolkit screen

The Toolkit screen consists of five sections. The default parameters for a terminal screen are 80 characters by 24 lines.

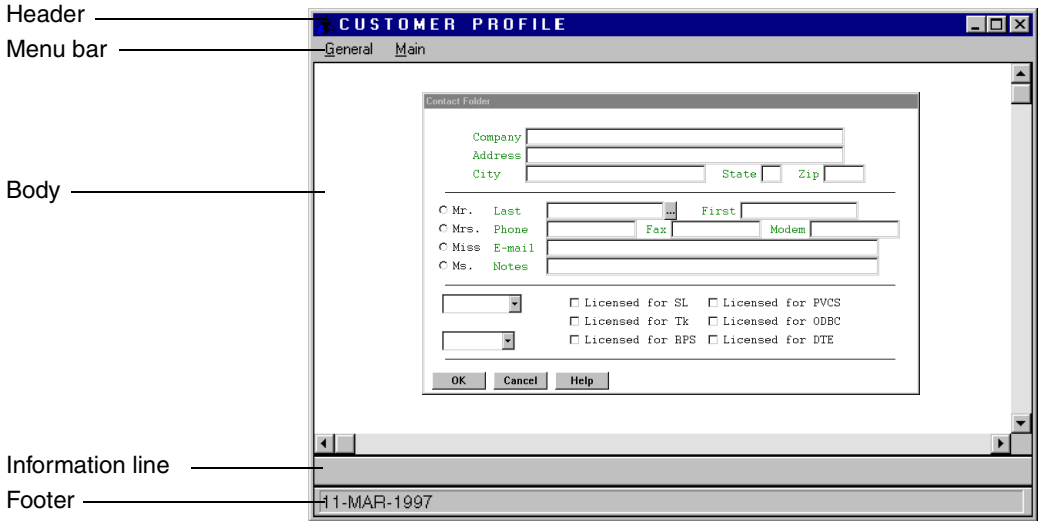



*Figure 6-1. A sample UI Toolkit screen.*

The five screen sections are

- **Header.** On UNIX and OpenVMS, the screen header can be zero to four lines. On Windows platforms it should be set to one line and becomes the window's "caption," or title. The application name, customer name, date, or other application-specific information is often displayed in the header.
- **Menu bar.** The menu bar is a single-line mandatory region that appears just below the header. All active menu columns are placed on the menu bar. When a menu column is placed on the menu bar, it is placed immediately to the right of the existing columns. When a menu column is removed, other columns shift to the left to fill in the void.

- **Body.** This region is where all of your windows, lists, and tabs are displayed. It occupies the entire area between the menu bar and the information line.
- **Information line.** This single-line section between the body and the footer is used to display dynamic, screen-specific information. For example, a brief description of the input required for the current input field might be displayed here.
- **Footer.** This is an optional section that is placed on the bottom of the screen, immediately beneath the information line. It can be zero to four lines. The footer is used in conjunction with the header and the information line to present information to your users. Like the header, the footer usually contains more static, application-specific information.

The following code is executed once, in the start-up routine. It initializes Toolkit and loads descriptive text into the header and the system date into the footer. No other Toolkit routine can be called until U_START has been called. As the chapter progresses, more code will be added to this start-up routine.

```
startup,
    xcall date(date)
    xcall u_start("cust_lib", 1, 1)      ;Initialize
    xcall e_sect("C U S T O M E R   P R O F I L E",
  &              D_HEADER, D_CENTER)     ;Place text in header
    xcall e_sect(date, D_FOOTER, D_LEFT) ;Place text in footer
    return
```

## Letting Toolkit manage your files

UI Toolkit needs to manage the files that are open on the system. Not only does it open your window library, it opens some files of its own as well. Toolkit keeps track of all your files and closes them automatically when you no longer need them. To take advantage of Toolkit's file management capabilities, and to avoid conflicts with the files Toolkit opens, you need to go through Toolkit to open and close your database files.

UI Toolkit's open routine is U_OPEN. U_OPEN opens your file on the next available channel and registers the file. When you are done with the file, you close it with the U_CLOSE subroutine. U_CLOSE closes the file and *deallocates* it (removes it from Toolkit memory).

Take a look at the sample code below. Notice the similarity between U_OPEN and Synergy Language's OPEN statement. (See the "Synergy Language Statements" chapter of your *Synergy Language Reference Manual*.) The first argument is the channel, but instead of specifying the channel, we pass a variable that is returned by U_OPEN with the next available channel. The second argument is an alpha literal that specifies the open mode. Next is the filename.

The primary difference between U_OPEN and OPEN is the way errors are handled. Because U_OPEN is an external subroutine, ONERROR or the I/O error list doesn't work. Instead, open errors are handled by passing an error variable in the sixth position. If an error occurs on the open, it is trapped and the error number is returned in the error variable. The loop we have set up doesn't

exit until the error variable returns false. If the error variable returns true, control returns to the top of the loop and a new file is created. All calls to U_OPEN should pass an error variable and test it before continuing.

```
do
  if (error)
    begin                       ;Create data file only if it doesn't exist
      xcall isamc("cdat", ^size(customer), 2, "start=1, length=45",
  &               "start=127, length=32, DUPS, MODIFY")
      xcall u_open(customer_ch, "u:i", "cdat",,, error)
    end
until (.not.error)
```

U_CLOSE is refreshingly simple. To close the file we just opened, we call U_CLOSE and pass the file's channel variable. If we had other files to close, we could pass their IDs here as well.

```
xcall u_close(customer_ch)
```

## Using event-style programming

To realize the full benefit of UI Toolkit, you should incorporate an "event-driven" style of programming. Event-style programming means that after all the components are in place, your program will call a Toolkit routine to get its user input. This routine will manage all interaction with the user until one of a set of special events causes it to exit. Your program then tests to determine the event that occurred and takes the appropriate action. Usually, after the action has taken place, your program will want to return to the Toolkit routine that generated the event to get further input from the user. Therefore most Toolkit routines that get input from the user are placed at the top of a loop.

## Including tools.def

Every UI Toolkit program must .INCLUDE "**WND:tools.def**." Contained in **tools.def** are the global variables that enable your application to communicate with the Toolkit routines. With these global variables, your programs can test for the event that caused a Toolkit routine to exit, determine the current state of a Toolkit component, instruct a Toolkit routine to perform a task or function in a particular way, specify to a Toolkit load routine the channel on which the primary window library is open, and pass or retrieve many other pieces of valuable information.

Also in **tools.def** are the defined values (with .DEFINE) used as arguments to the Toolkit routines. Most of these arguments are just a numerical value. If you examine **tools.def**, you can see most of the defined arguments consist of a descriptive name preceded by D_. For instance, the defined values for D_FOOTER, D_CENTER, and D_CLEAR are 3, 12, and 6. Using these definitions makes your code more self-documenting than if the corresponding numbers were used.

Consider the following:

```
xcall e_sect("Synergex", D_FOOTER, D_CENTER, D_CLEAR)
```

and

```
xcall e_sect("Synergex", 3, 12, 6)
```

Both of these statements do exactly the same thing: erase the footer region and center the text "Synergex" on the first line of the footer section. But the first example is much more expressive and far easier to maintain.

## Using variables for identifiers

Just as channel numbers are assigned to designate files and devices in your Synergy Language I/O statements, every UI Toolkit component (window, menu column, list, toolbar, or tab set) has a unique identifier. These identifiers are integer variables you define in the data division of your program. When a UI Toolkit component is loaded, the Toolkit load routine assigns the next available value to that component and returns it in the identifier you passed. From that point on, when your application needs to reference that component, you will specify its ID variable (*not* its name). For performance reasons, we suggest you use longword-aligned **i4** variables for these identifiers.

In our sample programs, you will notice that we try to convey information about the type of variable and its use by the name we choose. Defined values are uppercased and begin with **D_**, global variables begin with **g_**, subroutine arguments begin with **a_**, menu column IDs have a **_col** suffix, input window IDs an **_inp** suffix, list IDs a **_lst** suffix and so on. We encourage you to adopt a similar approach in the applications you develop.

# Managing Display Levels with Environment Processing

Among the most tedious tasks an application programmer must deal with is writing the code that captures the current screen so it can be restored after some interruption has taken place. In fact, many applications simply do not allow any external interference with the display during input. But no matter what kind of application you write, there will be situations in which you would like to alter some or all of the screen to display information or perform input, and then, when done, return the screen to its former state.

This ability to take a "snapshot" of the screen and then clean up and restore its original state at some future point is called environment processing. Instead of writing hundreds of lines of code each time you need to alter the screen, you simply make a single Toolkit subroutine call to enter a new environment. When you enter a new environment, a snapshot is taken of the screen, saving all of the characteristics of the previous environment. In the new environment, you can alter the display—add or remove screen components, open files, get input, modify screen sections. When you're done, another simple subroutine call exits the environment and restores the original screen.

What makes this capability so powerful is that you can "nest" this processing. If you need to save multiple layers, you simply take multiple snapshots. By encapsulating your code in callable routines with a single point of entry and a single point of exit, you can have display routines themselves save and restore the screen. It does not matter which routine calls a display routine; the routine starts by entering a new environment, thus preserving the existing screen. When the routine is done, it simply exits the environment, restoring the screen to its former state.

You cannot skip environments. Environment levels are not identified; they are managed on a stack and are exited in the reverse order from which they were created. The routines that are used to save and restore environments are the simplest routines in UI Toolkit.

To save a copy of the current screen and enter a new environment:

```
xcall e_enter
```

To exit an environment and restore the screen to its previous state:

```
xcall e_exit
```

These routines have no arguments.

## Local and global screen components

By default, when you load a window or column, it is *local* to the current environment. This means that any subsequent environment cannot delete it from Toolkit memory (although it can be removed from the screen), and when the environment that loaded it is exited, the window or column is deleted. If you want any environment to be able to delete a window or column, or if you don't want a window or column to be deleted when the environment that loaded it is exited, you can promote the window or column to *global* status by passing an argument to the Toolkit routine that loads it.

## See also

The "Environment Routines" chapter of your *UI Toolkit Reference Manual* for more information about environment processing.

# Managing Window Libraries to Store and Retrieve Display Components

Much of what you will do as you develop with UI Toolkit is to separate your procedural code from your display information. The procedural code will remain in your program source, but the display information will reside in script files. These script files will be processed into special ISAM files, called window libraries, that store the definitions of Toolkit screen components: input windows, text windows, selection windows, menu columns, and list classes. These definitions are in a special binary format that is optimized for processing by Toolkit.

## Benefits of window libraries

Window libraries enable you to make changes to the visual portion of your application without having to rebuild the associated code. Because the joining of these two parts is not done until the program is executed, you can change the visual portion of your application by changing your script files and rebuilding the associated window library.

A further benefit of window libraries is that you can have multiple libraries that use the same procedural code. Your basic application can be exactly the same, but you can have a different window library for each customer or department. For example, your procedural code could process a module that determines a user's security level. Based on this information, you could open a different window library that matches the security level associated with that user. One window library has an opening menu with 10 choices; another has an opening menu with 5 choices. Your application's procedural code does not require any change to restrict access to choices that are not on the menu: the choices simply do not exist for a user if the associated menu entries are not displayed.

## Specifying a window library

When you initialize UI Toolkit (using U_START), the first argument you specify is the name of your primary window library. The channel number on which this library file is opened is stored in the global variable **g_utlib**. Our examples contain many references to this variable.

However, your application can also have other window libraries open. To do so, simply use Toolkit's open routine, U_OPEN, to open the library in input mode. To access any secondary window library, simply pass the channel variable you passed to U_OPEN to the load routine.

# Creating Script Files and Window Libraries

Script files are text files created by Composer or a text editor. You can create window libraries using Composer, Script, or Proto. For information on how to use Composer, see chapter 4, "Designing Your User Interface." For information on how to use Script and Proto, refer to the "Script" and "Proto" chapters of your *UI Toolkit Reference Manual*.

## Creating and using menu columns, windows, and lists

### Creating menu columns

UI Toolkit provides windows-style pull-down menus in both Windows and non-Windows environments.

A menu column is a list of entries that pull down from the menu bar. The menu bar can accommodate many menu columns, although only one can be pulled down at any given time. The menu bar (and its associated columns) are activated by pressing ALT (on Windows) or CTRL+P (on UNIX and OpenVMS). Navigation through the columns is controlled by the arrow keys and by each entry's "quick-select" character. In Windows environments, full mouse support is enabled. In addition, when a menu column is not pulled down, special key *shortcuts* can be assigned to select a particular menu entry.

#### Step 1 - Defining menus

The first thing you must do to use Toolkit menu processing is define your menu columns. You can define menu columns with Composer or a text editor.

*Assigning shortcuts*
An optional menu shortcut enables the associated entry to be selected when the menu column is not pulled down (for example, while the user is doing input or viewing a list). Pressing a shortcut is the equivalent of pulling down a menu column, highlighting an entry, and pressing ENTER. For an entry's shortcut to be accessible to the user, its menu column must be "placed" (visible) on the menu bar. Valid shortcut keys include control, function, and arrow keys.

*Assigning quick-select characters*
When a menu column is pulled down, pressing an entry's quick-select character selects that entry if the quick-select character is unique. If the quick-select character is not unique, the entry will be highlighted. By default the quick-select character is the first character in the displayed text. Multiple entries can use the same quick-select character; the entries will be dealt with sequentially. Thus, in the sample menu script below, pressing an "a" (or "A") when the first menu is pulled down would position you to the "Abort" choice. Pressing "a" again would position you to the "About" choice.

*Using reserved entry names*
The menu entries described so far return as events to be processed by your program. But another class of entries is handled automatically by UI Toolkit, without ever exiting the Toolkit routine from which they were selected. These are called *reserved entries*. When any menu selection is made, Toolkit checks the entry name before returning to your program. If it recognizes the entry as one of the set of reserved entry names, Toolkit performs the associated task and then resumes processing without returning to your program.

These reserved entries are used to activate many of the standard functions of UI Toolkit, so you don't have to. Commands to control text editing, field navigation, context-sensitive help, and list navigation are among those enabled by reserved entries.

To take advantage of these capabilities,

1. Choose the functions you want to enable.
2. Include the appropriate reserved entry name in a menu column.
3. Place the column on the menu bar.

You don't have to write any code, you don't have to call any routines—just activate the feature by enabling your user to select it from the menu bar.

Below is a list of reserved prefixes to avoid when naming a menu entry that you want to return as an event to your program. A detailed list of each reserved entry and its associated function can be found in Appendix B of the *UI Toolkit Reference Manual*.

- C_ Composite window functions
- E_ Editing routine functions
- I_ Input routine functions
- O_ Key mapping or renditions
- S_ Selection window and list navigation functions
- T_ Text window viewing and display functions
- TS_ Tab set functions
- U_ Hot-entry processing

*Sample menu script*
The following commands define three menu columns. This code would reside in a script file.

```
.column general,  "General"
.entry o_help,    "Help",  key(F1)
.entry o_exit,    "Exit",  key(F4)
.entry o_abandon, "Abort", key(^A)
.entry g_about,   "About"
.end
```

```
.column main,     "Main"
.entry customer,  "Customers"
.entry vendor,    "Vendors"
.end

.column input,    "Input"
.entry i_prev,    "Previous field", key(UP)
.entry i_next,    "Next field",     key(DOWN)
.entry e_left,    "Move left",      key(LEFT),select(l)
.entry e_right,   "Move right",     key(RIGHT),select(r)
.end
```

The first three qualifiers in each menu column definition (the command, the name, and the text) are required and must be specified in order. In each .COLUMN line, the first qualifier after .COLUMN is the column name. This name is used in the program to load the column. The second qualifier (in quotation marks) is the header text. This text is visible on the menu bar when the column is *placed*, or made available to your user.

In the .ENTRY line, the qualifier in the second column is the entry name. When an entry is selected, the entry name is returned in uppercase to the program (using the Toolkit global variable **g_entnam**) so the program can determine the event that caused the Toolkit routine to exit. The third qualifier in the .ENTRY line contains the text that is displayed in the menu column. The entry options begin in the fourth column and can be specified in any order. The **key** keyword indicates the shortcut to be associated with the menu entry; **select** is the entry's quick-select character.

In most applications, a general menu column is available at all times to provide the basic functions the user can enter at any time (Help, Exit, Abandon, About, and any application-specific functions you choose). The first column in the sample script is an example of a general menu column.

The third menu column in our example (**input**) contains only reserved menu entries. When selected, these entries are processed automatically by UI Toolkit without returning to the program.

Note that each column definition is terminated with a .END. This is required.

*Compiling script files*

The next step is to add your definitions into a window library. See "Managing Window Libraries to Store and Retrieve Display Components" on page 6-10 for more information about window libraries.

### Step 2 - Loading the menus

To use menus in your program, you first have to load their definitions into UI Toolkit's memory using the M_LDCOL subroutine. This Toolkit routine locates the specified menu column in the window library and loads its definition into the Toolkit memory area. It then returns a value in the variable you have passed as the first argument. From this point on, every reference to this menu column is accomplished by referencing this variable. Like all other identifiers used by Toolkit, column and window IDs should be **i4** variables. The following code loads several menu columns:

```
startup,
    xcall date(date)
    xcall u_start("cust_lib", 1, 1)        ;Initialize--1 header line,
                                           ; 1 footer line
    xcall e_sect("C U S T O M E R   P R O F I L E",
  &              D_HEADER, D_CENTER)       ;Place text in header
    xcall e_sect(date, D_FOOTER, D_LEFT)   ;Place text in footer
    xcall m_ldcol(gen_col, g_utlib, "general",,, D_GLOBAL)
    xcall m_ldcol(options_col, g_utlib, "options")
    xcall m_ldcol(input_col, g_utlib, "input", D_NOPLC)
    xcall m_ldcol(select_col, g_utlib, "select", D_NOPLC)
    return
```

U_START places the channel for the primary window library (the first argument) in the global variable **g_utlib**. All of the columns loaded above reside in **cust_lib**. In this example, the **input** and **select** menu columns are loaded but not placed on the menu bar. Because this is the only place in the application that loads these columns, and this code is only executed once, there is no need to pass the optional search argument.

Notice too that the **general** column is promoted to global status. All global menu columns are placed on the menu bar to the left of local columns, regardless of their placement order. By making the **general** column global, we can also take advantage of UI Toolkit features that allow placement and removal of columns by their local or global status.

### Step 3 - Processing menus

The following sample code demonstrates basic menu processing. While placed menu columns and their entries are accessible in almost every UI Toolkit routine that receives input from the user, M_PROCESS is the Toolkit routine that forces the menu bar to process. Below is one method for giving your user a list of options at the start of an application. It assumes that the entry names returned by M_PROCESS match the name of the programs they activate.

```
do
  begin
    xcall m_process("[customer]")
    if (g_select)
      begin
        xcall e_enter
        xcall m_column(D_REMOVE, D_LOCAL)
        xcall xsubr(g_entnam)
        xcall e_exit
      end
  end
until (done_flg)
```

The code starts a loop that explicitly processes the menu bar until the user makes a selection. When the menu bar is first activated, the **customer** menu entry is highlighted. Until a selection is made, UI Toolkit controls the interface. When a selection is made, M_PROCESS exits and the program needs to determine the event that occurred.

We start by testing the global variable **g_select**. If this variable is true (in other words, nonzero) UI Toolkit is telling us that the reason it returned from M_PROCESS was that a menu entry was selected. Next we enter a new environment and prepare for the new program to be called by removing all local columns (those columns that are no longer valid now that a selection has been made). Since a new environment has been entered, whatever program gets called by the XSUBR subroutine can modify the Toolkit screen in any way necessary; these changes are reversed when the program exits and E_EXIT is called. When the user is done, he or she selects "Exit," which returns O_EXIT in **g_entnam**. Our external subroutine O_EXIT sets the global variable **done_flg** equal to true, and the program exits the loop.

Another way of processing a menu column is with a CASE or USING statement. Consider the following example:

```
repeat
  begin
    xcall m_process("[customer]")
    if (g_select)
      begin
        xcall e_enter
        xcall m_column(D_REMOVE, D_LOCAL)
        using (g_entnam) select
          ("CUSTOMER, VENDOR"), xcall customer
          ("OTHER"), xcall other
          ("O_EXIT"), exitloop
        endusing
        xcall e_exit
      end
  end
```

UI Toolkit uppercases the return values before moving them to **g_entnam**. If you specify your match values in lowercase, they will *not* match.

Each of the menu selections has a corresponding match value except **o_help**. **O_help** is a reserved entry that UI Toolkit processes directly. We will discuss help in "Implementing native Toolkit help" on page 6-36.

Note that the "OTHER" label has no corresponding menu entry. This entry can never be accessed by this program while the menus are defined as they are in this example. To enable this selection, a menu choice would have to be added to an existing menu column, or another menu column that contains this entry would need to be placed on the menu bar. This is a prime example of how you can distribute a complete application and only enable certain portions of the code, based solely on

the composition of the window libraries. If your customer wanted to pay for the additional features, you would simply have to provide a window library with a column that contains the additional entry—the customer would have access to that entry, and you wouldn't need to change any code.

While the CASE or USING approach may require a little more code, it is more self-explanatory: anybody looking at this code can see exactly what happens with each menu event. Of course, if the number of possible menu selections is large, the selection statement can become quite unwieldy, making the XSUBR approach more desirable. XSUBR is also more flexible, enabling you to create a generic menu processing routine and also to add an unlimited number of menu entries without modifying your code.

What has been discussed here is only a portion of the capabilities of UI Toolkit menu processing. Consult the "Menu Routines" chapter of your *UI Toolkit Reference Manual* for a comprehensive description of the Toolkit routines that affect menu processing. Additional options offer such features as menu creation at execution time, manipulation of menus in environments, and submenus. Because menu processing drives all Toolkit processing, a thorough understanding of how menus work is essential.

## Creating input windows

With an understanding of menu processing, we are ready for the next logical step. How do we display and retrieve information from our users? The answer: input windows.

To use a UI Toolkit input window, your perspective has to change. In your old style of programming, you had to deal with each field as a separate entity, taking care to position the cursor correctly, display the prompt, and then read the data. Once your program had the data, you had to decide if the data was valid and perhaps reformat it. If an input error was detected, you had lots of code to erase the screen section, reposition the cursor, acquire the input again, redo the edits—until you got acceptable data. Only when this process was complete could you move on to the next field.

With UI Toolkit input windows, you no longer have to think in terms of processing individual fields in a record. Instead, you deal with them as a group that is managed by Toolkit on your behalf.

Field information originates as an input window definition in a script file. The script file can either define field options locally, pull information from the repository, or do a combination of both. Since most fields in an input window have a corresponding field in the data area of your program, it makes sense to define this data once, in the repository, and reference the definition in both your source and script files. If you have input fields that will be used in more than one window, it's also helpful to use S/DE Repository to store field information (such as position, prompt, and validation). UI Toolkit's ability to extract information from Repository is one of its most useful features.

Most of the edits that you are used to making on input data can now be done automatically by UI Toolkit's input processor. For instance, you can specify that a particular field is decimal and must contain a value between 100 and 1000. If the user enters a value that is outside that range, or that is not decimal data, Toolkit will detect the error, display the appropriate message, erase the field, and reposition the cursor for input. For alpha fields, you can create a list of correct entries to be displayed for the user; the user will be able to select only from that list of valid options.

Of course, many other data validation options are available: you can require nonblank or nonzero input to a field, specify the help window for the field, provide text to be displayed on the information line when the field is processed, define a field's paint character, specify a field's display format string, identify routines to call when the user enters or leaves a field, and more. (For more information on arrive and leave methods, see "Using Methods" on page 6-45.)

With Repository you can define a field's characteristics once, and all input script references to that field definition are assured the same characteristics. If you need to change a field attribute (for instance, you have a department field that has an associated selection list and you want to add another department) all you need to do is add that department to the selection list in your repository and rebuild your window library. The next time your application is run, every instance of that field, on every screen in the application, will contain the additional department. You can also override any Repository characteristic (except the data type and size) in your script file.

### Navigating an input window

To handle interfield and intrafield movement in a non-Windows environment, UI Toolkit provides you with a number of reserved menu entries. The three primary reserved entry types are those that apply to every field (**I_** and some **E_**), those that apply to selection windows and lists (**S_**), and those that apply to text fields (**E_**). Define a separate menu column for each of these entry types.

Load (but don't place) each of these columns at start-up. When you call I_INPUT, pass the ID for the "input" column in the fourth argument position, the "select" column in the fifth position, and the "text" column as the sixth argument. I_INPUT will place and remove each column when appropriate, and when a selection is made from any of these columns, it will perform the designated task without returning to your program.

Because these tasks are handled automatically on Windows, UI Toolkit on Windows does not place these menu columns. If you plan to run your application as both Windows and non-Windows, don't mix reserved and nonreserved (to be returned as events to be processed by your application) menu entries in a column to be placed by I_INPUT. If you do, that column will not be placed on Windows, and events your program expects will never be generated.

### Step 1 - Defining input windows

Like all other UI Toolkit screen components, input windows are defined in script files and processed into window libraries. To define an input window, you can use Composer or a text editor. The following is a script file definition of an input window. In this example, all the field information is defined locally. This approach gives you the ability to view the entire structure of the input window, including the .FIELD options, at a glance.

```
.input customer, 14, 71
.placement 3, 5
.title "Customer Folder", color(3)
.line WndLin1, 65, pos(5, 4)
.line WndLin2, 65, pos(10, 4)
.line WndLin3, 65, pos(14, 4)
.repository_structure CUSTOMER
```

```
.field cmpny, a45, prompt("Company "), pos(2, 8), fpos(2, 16), break, -
     drill_method("cmpny_drill"), hyperlink_method("hyperlink")
.field address, a45, prompt("Address "), pos(3, 8), fpos(3, 16), -
     hyperlink_method("hyperlink")
.field city, a25, prompt("City "), pos(4, 8), fpos(4, 16), -
     hyperlink_method("hyperlink")
.field state, a2, prompt("State "), pos(4, 44), fpos(4, 50), -
     hyperlink_method("hyperlink")
.field zip, d5, prompt("Zip "), pos(4, 55), fpos(4, 59), -
     hyperlink_method("hyperlink")
.field a4, salutation, fpos(6, 3), -
     select(0, 0, 4, "Mr. ", "Mrs.", "Miss","Ms. "), radio, -
     select(0,0,1, "Mr.", "Mrs.", "Miss", "Ms.", -
     hyperlink_method("hyperlink")
.field lname, a16, prompt("Last "),  pos(6, 11), fpos(6, 19), -
     help("h_lname"), drill_method("contact_drill"), -
     hyperlink_method("hyperlink")
.field fname, a16, prompt("First "), pos(6, 41), fpos(6, 47), -
     hyperlink_method("hyperlink")
.field phone, d10, prompt("Phone "), pos(7, 11), fpos(7, 19), -
     hyperlink_method("hyperlink"), format("XXX/XXX-XXXX")
.field fax, d10, prompt("Fax "), pos(7, 33), fpos(7, 37), -
    hyperlink_method("hyperlink"), format("XXX/XXX-XXXX")
.field modem, d10, prompt("Modem "), pos(7, 51), fpos(7, 57), -
     hyperlink_method("hyperlink"), format("XXX/XXX-XXXX")
.field email, a47, prompt("E-mail "), pos(8, 11), fpos(8, 19), -
     hyperlink_method("hyperlink")
.field notes, a47, prompt("Notes "), pos(9, 11), fpos(9, 19), -
     hyperlink_method("hyperlink")
.field type, a10, fpos(11, 4), hyperlink_method("hyperlink")
.field status, a10, fpos(13, 4), hyperlink_method("hyperlink")
.field sl, d1, prompt("Licensed for SL "), fpos(11, 22), checkbox
.field tk, d1, prompt("Licensed for Tk "),  fpos(12, 22), checkbox
.field rps, d1, prompt("Licensed for RPS "),  fpos(13, 22), checkbox
.field pvcs, d1, prompt("Licensed for PVCS "), fpos(11, 42), checkbox
.field odbc, d1, prompt("Licensed for ODBC "), fpos(12, 42), checkbox
.field dte, d1, prompt("Licensed for DTE"), fpos(13, 42), checkbox
.button I_OK, text("OK")
.button O_ABANDON, text("Cancel")
.button O_HELP, text("Help")
.button_set bottom(3)
.set customer, CUSTOMER, cmpny, address, city, state, zip, salutation, -
     lname, fname, phone, fax, modem, email, notes, type, status, sl, -
     tk, rps, pvcs, odbc, dte
.set nokey, CUSTOMER, address, city, state, zip, salutation, lname, -
     fname, phone, fax, modem, email, notes, type, status, sl, tk, rps -
     pvcs, odbc, dte
.end
```

Take a look at the input window below. It's the same input window, but in this example its field information is pulled from the repository. The advantage to defining your fields in Repository is that all the qualifiers for each field are defined only once, no matter how many times the field is referenced. There is no need for the .STRUCTURE command, because the structure information is part of the repository definition. If you include the repository definition in your program as well (as you should), a match between the data in your program and the fields in your window definition is ensured: the input processor will automatically know where to put the data for each field, regardless of the order specified or gaps left by the exclusion of any fields. A downside to the repository approach is that to view field information, you must select the structure in Repository and look at each field definition separately.

Notice in the example that the **lname** and **fname** fields have a **required** qualifier. Any qualifier not specified in the repository can be added in the script file. If any qualifiers conflict, the one in the script file takes precedence. You can also turn off repository qualifiers in the script file.

```
.input customer, 14, 71
.placement 3, 5
.title "Customer Folder", color(3)
.line WndLin1, 65, pos(5, 4)
.line WndLin2, 65, pos(10, 4)
.line WndLin3, 65, pos(14, 4)
.repository_structure CUSTOMER
.field cmpny
.field address
.field city
.field state
.field zip
.field salutation
.field lname, required
.field fname, required
.field phone
.field fax
.field modem
.field email
.field notes
.field type
.field status
.field sl
.field tk
.field rps
.field pvcs
.field odbc
.field dte
.button I_OK, text("OK")
.button O_ABANDON, text("Cancel")
.button O_HELP, text("Help")
.button_set bottom(3)
```

```
.set customer, CUSTOMER, cmpny, address, city, state, zip, salutation, -
     lname, fname, phone, fax, modem, email, notes, type, status, sl, -
     tk, rps, pvcs, odbc, dte
.set nokey, CUSTOMER, address, city, state, zip, salutation, -
     lname, fname, phone, fax, modem, email, notes, type, status, sl, -
     tk, rps, pvcs, odbc, dte
.end
```

Note that both input window definition methods have some things in common. Anything that does not pertain to a field definition is specified entirely in the script file. A default screen position is set, a title is specified, and lines are drawn to organize the screen into logical sections. The .SET commands create input sets: groups of fields to be processed or accessed as a unit. And on Windows systems, three buttons, “OK,” “Cancel,” and “Help” will be displayed. The .BUTTON_SET command specifies that these buttons will appear horizontally at the bottom of the screen (three buttons per row).

### Step 2 - Loading input windows

Just like we had to load our menu columns with UI Toolkit, we also must load our input windows. The routine that does this is I_LDINP.

```
xcall i_ldinp(customer_inp, g_utlib, "customer", D_NOPLC)
```

Notice in the sample that the definition of the window is taken from the default window library (the file open on the channel contained in **g_utlib**), and that we are not placing it on the screen at this time (D_NOPLC). It is a common practice to load all windows and menu columns in your initialization code and then place them only when needed.

### Step 3 - Displaying an existing record

To display read-only data to a user, use I_DISPLAY to “display” or load the data into the window. Then the U_POPUP routine can be used to hold the information on the screen until the user presses ENTER.

```
display_record,
   xcall e_enter                              ;So we can restore the screen
   xcall u_window(D_REMOVE, D_ALL, D_PLACE, customer_inp)
   xcall e_sect("View customer info", D_INFO, D_CLEAR, D_LEFT)
   xcall i_display(customer_inp,, customer)   ;Fill the window with data
   xcall u_popup(g_utlib, "customer")         ;Freeze the screen
   xcall e_exit                               ;Then restore the screen
   return                                     ;Return to where we were
```

In this example, a new environment is entered to enable us to reverse screen changes when we are done. Next, U_WINDOW is called to remove all other windows and place the customer window. I_DISPLAY is then called to load the window with data. Once the window contains the data to display, U_POPUP is called to update and freeze the screen. When the user presses ENTER, the new environment is exited, returning the screen to the state it was in when **display_record** was called.

### Step 4 - Adding a new record

An input window that is used to get user input is processed in a loop. Consider this example:

```
process_window,
    xcall e_enter
    xcall u_window(D_PLACE, customer_inp)     ;Place at default position
    do
      begin                                   ;Begin input loop
        xcall i_input(customer_inp, "customer", customer, input_col,
  &                   select_col, text_col)
        if (g_setsts) then                    ;Menu entry or break field
          begin
            if (g_select) then
              call menu_choices               ;Menu selection
            else
              call break_fields               ;Break field
          end
        else
          call record_complete                ;Input set complete
      end                                     ;End input loop
    until done
    clear done
    xcall e_exit
    return

break_fields,
    using (g_fldnam) select
      ("CMPNY"), begin                           ;Check after each field
                   if (C_LNAME.and.C_FNAME)      ;Is key complete?
                     call test_for_duplicates
                     if (duplicate) then
                       call init                 ;Initialize for next input
                 end
    endusing
    return

init,
    xcall u_message(%atrim(cmpny)+" already exists")
    xcall i_init(customer_inp, "customer", customer)
    return

menu_choices,
    using (g_entnam) select
      ("O_EXIT","O_ABANDON"), done = TRUE
      ("G_ABOUT"),            xcall u_about(TITLE, VERSION, %datecompiled)
    endusing
    return
```

```
record_complete,
    store(customer_ch, customer) [$ERR_NODUPS=store_error]
    call init
    return
```

Here the input processor has control of all input until one of three events occur:

- The user makes a menu selection.
- A “break” field is processed.
- Every field in the input set has been input to.

After the call to I_INPUT, we test for one of these three events and call an internal subroutine to handle the event.

The key to understanding input processing is understanding the structure of the input processing loop and the tests performed to determine the exit event. As we did in menu processing, we use global variables provided by UI Toolkit to determine the event that caused I_INPUT to exit. In our example we first examine **g_setsts** to find out if input processing is incomplete. If the value is true (nonzero), we know input processing is incomplete—either a menu selection or a break has occurred. Next we test **g_select**. If it is true, we know the user made a menu selection. A USING statement determines the menu selection that was made. If **g_select** is false, the exit event was a break field; this time the USING statement determines the field that activated the break. If **g_setsts** is false, no further processing needs to be done on this input set. We store the completed record and initialize the window in preparation for the next record.

When I_INPUT exits, it retains the *context* for the next time it is called. The context is simply the next field to process. By default, it is always the field after the field that caused I_INPUT to exit. So after a break field is processed or a menu selection is made, when the program returns to the top of the loop and calls I_INPUT again, processing resumes right where it left off. I_NEXT (see the example below) changes the input context, and I_INIT resets the input window to its initialized state. The context for an initialized window is the first field in the input set.

See step 5 for another example of how to add a record.

### Step 5 - Modifying an existing record

To use your input window to modify an existing record, load the data into the window with I_DISPLAY. Use I_NEXT to set the context (the input field to be processed with the next call to I_INPUT).

```
process,
    xcall e_enter
    set = "customer"
    do
      begin
        xcall i_input(customer_inp, set, customer, input_col, select_col)
        if (g_setsts) then       ;Check for incomplete input processing
          if (g_select) then     ;Menu processing
```

```
            xcall xsubr(g_entnam) ;Entry name matches subroutine name
          else
            begin                 ;Break processing
              using (g_fldnam) select
                ("CMPNY"), call dup_chck
              endusing
            end                   ;Break processing
        else
          begin                   ;Input set complete
            if (load_flg) then    ;Either MODIFY mode...
              begin
                read(customer_ch, cmpny, cmpny) [$ERR_TOOBIG=cont]

cont,
                write(customer_ch, customer)
                clear load_flg
              end
            else                  ;... or ADD mode
              using (set) select
                ("customer"), store(customer_ch, customer) [ERR=updterr]
                ("nokey")   , write(customer_ch, customer) [ERR=updterr]
              endusing
            if (FALSE)
              begin               ;Store failed

updterr,
                xcall u_message("Error updating customer file")
              end
            xcall init            ;Initialize window, enable cmpny field
          end                     ;Input set complete
      end
    until (done_flg)
    xcall e_exit
    return

dup_chk,
    find(customer_ch, customer, cmpny) [$ERR_KEYNOT=newrec,
  &      $ERR_EOF=newrec]
    reads(customer_ch, customer)
    xcall i_display(customer_inp, set="nokey", customer)
    xcall i_next(customer_inp, set, "*FRST*")

newrec,
    return
```

In this example we take advantage of multiple input sets to either add or modify data. If we determine in the **dup_chk** internal subroutine that the record exists, we read the data and load it into the input window with I_DISPLAY. Notice that we change the input set from **customer** to **nokey**. This is because we will be modifying an existing record and don't want the user to modify the primary key. If you look in the script definition for the **customer** window, you'll see that the **nokey** input set contains all fields except the primary key, **cmpny**. Because I_DISPLAY flags every field as having input entered in it, I_NEXT is called to set the context to the first field in the **nokey** input set. When **dup_chk** returns to the input processing loop and re-executes I_INPUT, the input window will contain the data to modify, and the user will not be able to access the **cmpny** field.

If the FIND statement fails to find a matching record, it skips the code that updates the input window and returns to the input loop to continue "add" processing.

## Creating selection windows

Selection windows are a powerful way to present a set of valid options to your user. If you specify a list of values for a field (in either its script or repository definition), when that field is processed, a selection window containing those choices appears. Your user can only select from those choices.

If several of your input windows contain fields that share a set of valid choices, you can create a single selection window and associate it with each of those fields. Thus, you maintain the list of choices in only one place, and by changing just that window, every field that uses the window is changed.

### Step 1 - Defining selection windows

Selection windows are quite simple to create and maintain. You can use Composer or a text editor to create selection windows. (Composer uses its own text editor to create selection windows.) The following sample code defines two selection windows:

```
.select yes_no
.placement 2, 2
.border on, color(1)
.item "Yes"
.item "no"
.end

.select marital_stat, 2
.placement 2, 2
.border on, color(1)
.item "Married"
.item "Divorced"
.item "Widowed"
.item "Single"
.end
```

The first selection window is a generic window for yes or no answers. The choices are displayed vertically. In most applications, you would also create a **no_yes** selection window with the items reversed. This is because the default choice is the first one listed; assigning the appropriate selection window to your fields (with the **selwnd** .FIELD qualifier) makes your application more user-friendly. The second window is a marital status list. It has four entries. The "2" on the .SELECT command is the number of rows to display; the entries in this column are split into two two-row columns.

### Step 2 - Processing selection windows

To move about in your selection windows in a non-Windows environment (it is handled automatically on Windows), create a menu column that contains the selection window reserved menu entries (**S_**). In most cases, you will associate each entry with a shortcut. (The arrow keys are a logical choice.) If you pass the column ID as the fifth argument to I_INPUT**,** the column will be placed whenever a selection field is processed and removed as soon as that selection field is exited. The sample below uses the reserved entries and associates them with the arrow key shortcuts. It is to be loaded (but not placed) at start-up.

```
.column sel_col, "Selection keys"
.entry s_up, "Up", key(UP)
.entry s_down, "Down", key(DOWN)
.entry s_right, "Right", key(RIGHT)
.entry s_left, "Left", key(LEFT)
.end
```

### Dynamic selection windows

We have only discussed static selection windows in this section. UI Toolkit also provides routines that enable you to build selection windows at runtime. This feature enables your application to create and modify selection windows based on factors that are not determined until the program executes. For example, selections can be loaded from a file, based on security or license status. See the "Selection Routines" chapter in your *UI Toolkit Reference Manual* for more information.

## Creating text windows

One of the most difficult things to code is an editor for text entry. Many applications require editing capabilities, whether it be to allow entry of comments on a purchase order, special instructions on a delivery order, or justification on a General Ledger entry. Some applications address the problem by calling a system utility to perform the editing. But what happens when you move to another platform and that utility is not available? Or you upgrade your system and the utility you previously used is not yet available on the new version? Your application doesn't work, or its capabilities are severely diminished. To address this problem, and to give your application true platform independence, UI Toolkit provides text-editing capabilities.

### Step 1 - Define the functions

UI Toolkit enables you to edit or simply view text. Both capabilities are handled via text windows and reserved menu entries. Edit functions are activated by the **E_** reserved entries, and the text view functions (scrolling) by the **T_** reserved entries. Each set of entries should reside in its own column, with a unique shortcut key assigned to each entry. This script shows typical edit and view columns:

```
.column      textedit,     "Edit text"
.entry       e_left,       "Move left",              Key(left)
.entry       e_right,      "Move right",             Key(right)
.entry       e_emov,       "End of line",            Key(end)
.entry       e_bmov,       "Beginning of line",      Key(home)
.entry       e_xup,        "Move up",                Key(up)
.entry       e_xdown,      "Move down",              Key(down)
.line
.entry       e_cdel,       "Delete character",       Key(del)
.entry       e_lclr,       "Clear line",             Key(^X)
.entry       e_edel,       "Clear to EOL",           Key(^E)
.entry       e_ilin,       "Insert a line",          Key(ins)
.entry       e_join,       "Join lines",             Key(^J)
.end

.column      textview,     "View text"
.entry       t_top,        "Top of window",          Key(^T)
.entry       t_bottom,     "Bottom of window",       Key(^B)
.entry       t_lside,      "Left side of window",    Key(^L)
.entry       t_rside,      "Right side of window",   Key(^R)
.line
.entry       t_pagup,      "Scroll up page",         Key(prev)
.entry       t_pagdwn,     "Scroll down page",       Key(next)
.entry       t_scrup,      "Scroll up half page",    Key(f8)
.entry       t_scrdwn,     "Scroll down half page",  Key(f7)
.end
```

### Step 2 - Define the text window

The text window should be large enough to contain the text to be viewed or edited. If the window is too large for the screen, you must specify a display area that fits the screen. UI Toolkit will use the reserved entries your user selects to manage the scrolling of the display. This script window definition can be created in Composer or a text editor.

```
.window text, 99, 73
.display_area 1, 1, 20, 73
.placement 2, 2
.end
```

The sample above defines a window that is 99 lines long and 73 characters wide. The display area is 20 by 73, and starts in row 1, column 1 of the window. By default, the window will be placed at row 2 column 2 of the display region. U_WINDOW can be used to change the default placement.

### Step 3 - Load the windows and menu columns

Just as we have done with all of our other Toolkit objects, we need to load our window and our menu columns. The routine used to load a display (noninput) window is U_LDWND. Notice in the example below that neither the window nor the columns are placed. The window is placed when it is time to use it, and the UI Toolkit text routine places and removes the columns automatically.

```
xcall u_ldwnd(text_win, g_utlib, "text", D_NOPLC)
xcall m_ldcol(view_col, g_utlib, "textview", D_NOPLC)
xcall m_ldcol(edit_col, g_utlib, "textedit", D_NOPLC)
```

### Step 4 - Convert the window

Once the window is loaded, it must be converted into a text window. T_SETUP establishes the controls for the display of the window and the default behavior for subsequent data that is loaded into the window. Display characteristics such as margin settings, paragraph treatment, display color and renditions, text wrapping, and display size can all be set with T_SETUP.

For our example, the window is placed as specified in the script file. We use the T_SETUP defaults and clear the window of any text that might have remained from the previous edit.

```
xcall e_enter
u_window(D_PLACE, text_winid)           ;Place at default position
xcall t_setup(text_winid, D_INIT)       ;Initialize the window
```

### Step 5 - Add data to the text window

To edit the text window, we place the call to T_EDIT in a loop. Exit from the loop is controlled by menu entries in a column placed before the call to T_EDIT (a column that doesn't contain any reserved menu entries). We allow T_EDIT to load our reserved entry menu columns for us. The example takes all of the default text processing modes (start at row 1, column 1; set insert mode; edit in a forward direction; allow lowercase input). By passing the variables in the row, column, insert, and case positions, we save the exit value for each. When the program returns to the top of the loop and calls T_EDIT, editing resumes in exactly the same place, in the same mode.

```
repeat
  begin
    xcall t_edit(text_win, row, col, insert, dir, case, edit_col,
  &              view_col,, "text_help")
    if (g_select)
      case g_entnam of
        begincase
"SAVE":   begin                         ;Save and resume
            call write_data
            exitloop
          end
"O_ABANDON": exitloop
        endcase
  end
```

```
xcall e_exit
return
```

### Step 6 - View the contents of the text window

Use T_VIEW to enable the user to scroll through the contents of a text window without editing it. The example below does not put T_VIEW in a loop because the only event that causes it to exit is "Exit." When the user selects "Exit," there is no reason to re-enter. Scrolling through the window is handled by the reserved menu entries contained in the **view_col** column.

```
view_text,
    xcall e_enter
    u_window(D_PLACE, text_winid, 1, 1)
    xcall t_view(text_win, view_col, "view_help")
    xcall e_exit
    return
```

*Extracting data from a text window*
Once the user has finished editing, we can do one of two things to save the input:

- Store the contents of the window into a window library.
- Extract the information from the window into our program's data area so it can be processed in some manner.

Our example extracts the data to the program. To store the window and its contents to a window library, see the U_SAVE subroutine in the "Utility Routines" chapter of the *UI Toolkit Reference Manual*.

### Step 7 - Define a data structure to hold the window contents

The first thing needed to extract the data from a text window is an alpha array with at least as many elements as there are rows in the text window. The size of each element should match the number of columns in the window. In the example, we allocate an alpha array whose dimension is based on a compiler definition (NUMLINES). We use a .DEFINE so we can use the same value as an upper boundary when we unload the window data into the array. If at some point in the future the text window needs to change size, we only have to change the value in the .DEFINE; all references to this value will remain synchronized.

```
.define NUMLINES,       99              ;Number of lines in text window
record
    textline        ,[NUMLINES]a73
```

### Step 8 - Unload the data

To unload the data, we call the UI Toolkit routine T_GETLIN. Our example routine transfers the data from the text window into the **textline** array, starting with line 1 and continuing through line 99 (NUMLINES). Because the user may not have filled the entire window with text, it might be beneficial to know exactly how many lines were retrieved; this value is returned in the **lines_input** variable. This variable is used as the upper boundary in the routine that stores the data.

```
write_data,
    xcall t_getlin(text_win, textline, NUMLINES, start_line=1,
  &                lines_input)
    for ivar from 1 thru returned           ;Now store the data records
      begin
        entry_number = hold_entry
        entry_line = ivar
        entry_text = textline[ivar]
        store(text_ch, diary)
      end
    return
```

### See also

The "Text Routines" chapter in your *UI Toolkit Reference Manual* for more information about other text processing features that include the ability to load text into a window and store data with the window in a window library.

## Processing lists

A *list* is a virtually unlimited collection of identically formatted items, its size limited only by the disk space available. In its most basic form, a list is a visual representation of a file. But a list doesn't have to derive its contents from a single file—you can construct the items in a list of data from any source or any combination of sources. But, like the records in a file, while the contents of each item are different, each item is structured the same.

An input window defines the structure of a single list item. Usually you create this input window for the sole purpose of associating it with a list. To display the list on the screen, the list processor makes multiple copies of this input window, each with its own data.

Of course, you may want to do some things a little differently for your list input windows. If you want your application to be portable to Windows platforms and want to use multiline list items, you will need to use the ActiveX list control when processing your list. With a UI Toolkit list, you are restricted to single-line list items (and thus a single-line list input window). In most cases, your list will have a header, so having prompts in your window will not make much sense. Also, if you pull the field definitions from your repository and the repository contains default values for *pos* and *fpos*, make sure you override them in the script definition.

In your script file you will also define the *list class*. While an input window defines the structure of a list's items, the list class defines the characteristics for the entire list. Things like the number of items visible at one time, the list's position on the screen, any *methods* (special subroutines called by the list processing routines) associated with the list, and the number of header and footer lines are all part of the list class.

When the list processor is called, it tries to perform the request your program passed to it. If this request requires an item that hasn't been loaded onto the list, the list processor calls your *load method*. A load method is a subroutine written by you that the list processor calls each time it needs a new item. Once an item is loaded, the list processor manages that item itself, storing its information in its own temporary files. The load method is called only once per item, regardless of the number of times that item is accessed.

A list item consists of two components:

- The display component, which is the formatted information as it is displayed on the screen
- The data component, which is data that you associate with the item

In the simplest case, the display component matches the data component, but there is no requirement that they be related. Your load method should load the list's display component into its input window and the data component into its data area.

### Step 1 - Define your display window

Our sample window contains a field that will be loaded (as the title indicates) with the company and contact names. The data to be loaded into this field is constructed in the load method, from the **company**, **fname**, and **lname** fields.

```
.input cstlst_inp, 1, 64
.placement 2, 2
.title "Company, Contact"
.field display, a64, fpos(1,1)
.end
```

### Step 2 - Define your list class

As you can see, there isn't much to a list class definition. In the sample below, we define a page as containing five items, reserve one line for the header, and specify the load method. The list will be placed at row 1, column 15.

```
.listclass customer_list, items(5), -
                          headers(1),  -
                          load_method("load_cust")
.placement 1, 15
.end
```

### Step 3 - Create the load method

Because your application does not call the load method directly, you do not control the arguments passed to it—you must write the routine to conform to the arguments that the list processor passes. The arguments must match the format below exactly.

Communication between the subroutine and the list processor is accomplished through the request argument (**a_request**). A list can be *expanded* (a new item added) at the top, at the bottom, or at the top or bottom, depending on settings specified when the list is loaded from the window library. If the list is being loaded in both directions, your load method will check the request argument to determine whether the next new item is to be added to the top or the bottom.

Do not depend on the order in which items are requested when two-way loading is enabled, because the user can change the order in which items are loaded for any given list.

In the example below, each new item is added to the bottom, so you don't need to check **a_request**.

Using a global variable for the input channel (the file was opened at start-up in the main routine), the load method reads sequentially through **customer.ism** from top to bottom (one record per call), constructs the display string, and calls I_DISPLAY to update the list's input window with the display for each new item.

In addition to the visible portion, data must also be associated with each item. **A_data** is the data area passed to the list processor (L_SELECT or L_PROCESS) and the list creation routine (L_CREATE). Once the item's data has been returned by the load method to the list processor, your program can use it to search the list for a specific item, determine the current list item, perform file look-ups, load an input window for modification or viewing—just to name a few things. It is important that you pass the data area to be associated with the list to all the list routines and that you load it with the appropriate data in your load method.

When the load method READS encounters end of file, it skips the load logic and sets **a_request** equal to **D_LEOF**. This tells the list processor that no more items are to be added to the list. From this point on, the list processor will not call the load method again, and instead will manage all list items itself.

```
subroutine load_customer
    a_listid        ,n     ;List ID
    a_request       ,n     ;RETURNED - LP communication variable
    a_data          ,a     ;Data area (customer record)
    a_inpid         ,n     ;Input window ID
    a_disabled      ,n     ;Flag for disabling an entry
    a_itemindex     ,n     ;The item's ordinal position in the list

.include "WND:tools.def"
.include "ids.gbl"
```

```
record
    display ,a65

proc
    begin
      reads(custload_ch, customer, eof)
      xcall i_display(a_inpid,,(display=%atrim(cmpny)+",
  &                   "+%atrim(fname)+" "+lname))
      a_data = customer
      exit

eof,
      xcall i_init(list_inp)
      a_req = D_LEOF
    end
    xreturn
endsubroutine
```

Because load methods are just external subroutines, your application program can use all of the power of Synergy Language to select or construct the items to add to the list. If you choose not to specify the load method in your .LISTCLASS definition, or if you want to use a different load method, the L_METHOD subroutine enables you to specify or change the load method at runtime.

If your load method loads items sequentially from a file, the record pointer *must* be on the next record to be loaded. If the load method retrieves data on the same channel used to update the database file, there is no assurance that the pointer is on the correct record. Therefore, you must open the input file on a channel dedicated to the load method. No other I/O can occur on this channel. If the list is being expanded in both directions, the file must be opened twice: once for the forward READS, and once for the reverse READS. In other words, a channel must be dedicated to the list load method *for each load direction***.**

### Step 4 - Load your list input window and selection column

Before you can associate your list input window with a list, the window must be loaded. And, like other UI Toolkit routines that get input from the user, the list processor's selection routine recognizes a set of reserved menu entries. The following statements to load both of these list components appear in the program's start-up routine. Make sure you load your list input window and selection column with the **D_NOPLC** argument—the input window and selection column will be placed and removed with the list, automatically.

```
xcall m_ldcol(lsel_col, g_utlib, "list_sel", D_NOPLC)
xcall i_ldinp(customer_lst_inp, g_utlib, "cont_l_inp", D_NOPLC)
```

### Step 5 - Create the list

All of the components are in place and ready to be assembled. We create our list with a call to the UI Toolkit routine L_CREATE. Although the syntax is different, L_CREATE can be compared to Toolkit's load routines, in that it goes out to the window library, finds the specified definition, and loads it into memory.

The creation of the list establishes the controls that will be used by UI Toolkit to manage the list. This is where the list's display and data components are assigned—the second argument to L_CREATE is the ID of the list input window, and the third argument is the data area to be associated with the list. Each of these variables is passed by the list processor as arguments to your load method.

```
xcall l_create(customer_lst, customer_lst_inp, customer, g_utlib,
  &            "listclass",,, D_NOPLC)
```

This sample code creates a simple list that will be loaded from top to bottom (the default). The list is loaded into UI Toolkit memory at start-up but is not placed on the screen. The **customer** record is established as its data area—it must be passed in all calls to the list processor as well.

L_CREATE also has additional arguments that control such things as the manner in which the list is loaded and its placement on the screen.

### Step 6 - Process the list

The template for processing a list is quite similar to that for processing other UI Toolkit objects. A loop structure calls the Toolkit processing routine, L_SELECT. This routine processes the list selection-style. The first time L_SELECT is called, the list processor determines that it needs an item. It calls the load method, which returns with an item to be added to the list. The list processor adds the item, and because the list class says there are five items on a page, it repeats this step four more times.

As soon as a page has been filled, L_SELECT puts a selection window on the screen. The user can move around through all the visible items (using the shortcuts associated with the reserved menu entries, or on Windows, using the mouse and arrow keys). With any request for an item not yet loaded—for example, if the user presses the shortcut to move down a page—the list processor once again calls the load method until the request has been filled. A request for an item (or page of items) that's already loaded does not initiate a call to the load method. Instead the list processor retrieves the item (or items) from the *file stack*, a temporary file management system the list processor uses to manage the list.

```
repeat
  begin
    xcall l_select(customer_lst, request, customer,,,,,, lsel_col)
    using (g_entnam) select
      ("G_ABOUT"),  xcall u_about(TITLE, VERSION, %datecompiled)
      ("O_EXIT"),   exitloop
      ("P_REMOVE"), call remove_entry
```

```
      ("P_MODIFY"), call modify_entry
      ("P_ADD"),    call add_entry
      (),           if (request)              ;No menu entry
                      call process_request    ;Process list request
                    else
                      call display_entry      ;Display selection
    endusing
  end
return
```

L_SELECT exits when a menu selection is made or when the user selects an item by highlighting it and pressing ENTER. Though it isn't displayed here, the **display_entry** routine determines the item selected by checking the data area passed to L_SELECT (as the third argument). It is *essential* that you pass to L_SELECT the same data area specified in the third argument to L_CREATE. If you don't, there is no way to determine which item was selected.

### See also

The "List Routines" chapter in your *UI Toolkit Reference Manual*. List processing is truly the most powerful display tool in UI Toolkit. The example here demonstrates only the most basic aspects. The possibilities for managing the display of data, from doing input directly into an existing list to advanced search capabilities, are extensive.

# Implementing Online Help

Toolkit has routines that support three types of online help: native Toolkit help, HTML Help (**.chm**), and WinHelp (which is now obsolete). Native Toolkit help is the default mechanism for displaying context-sensitive help. It uses Toolkit windows to display autonomous help topics (as opposed to a help system). To implement native Toolkit help, you'll associate help windows with fields and/or some routines, and you'll use the O_HELP menu entry to activate help. See "Implementing native Toolkit help" below.

With HTML Help and WinHelp, on the other hand, your help system can include a table of contents, index, and search (as well as different types of windows). Once the help system is open, users can use these and other features to navigate to other help topics. To implement HTML Help or WinHelp, you'll call %U_HTMLHELP or %U_WINHELP. These routines wrap most of the functionality of the Microsoft HTML Help API and WinHelp API. The state of the help system when it opens depends on the options you pass in the call and the features available in the help system (table of contents, index, etc.).

No matter which of these help formats you use, you can use Toolkit's built-in context-sensitive help features to associate a field and, in some cases, a routine with a help topic. These features include a replaceable help method, the O_HELP reserved menu entry, and context IDs for fields and some routines (as discussed in "Implementing native Toolkit help" below).

One way to implement HTML Help or WinHelp is to register your own Toolkit help method, and have that help method make the %U_HTMLHELP or %U_WINHELP call. If you do this, the O_HELP menu entry invokes your help method, and context IDs you've defined are automatically passed to the method. For example, with the following code an, HTML Help system (**myhelp.chm**) opens when the user presses F1 or selects Help from the menu. A context ID may be passed to the help method, so the help method has an argument that accepts a context ID. (For information on the cases in which a context ID is passed to the method, see EHELP_METHOD in the "Environment Routines" chapter of the *UI Toolkit Reference Manual*.) If a context ID is passed to the subroutine, the %U_HTMLHELP call displays the help topic for the context ID. If no context ID is passed (or if there's no help topic for the context ID), the help system opens with the table of contents displayed.

```
proc

xcall e_method(D_METH_HELP, "help_method") ;This registers the help
                                           ; method defined below.
.
.
.
xcall mb_entry(cont, "O_HELP", "Help", F1_KEY)
.
.
.
xcall ib_field(build_id, 'select1', D_FLD_SIZE, 10, D_FLD_TYPE,
```

```
&               D_DECIMAL, D_FLD_POS, 2, 2, D_FLD_PROMPT,
&               "Enter a number:", D_FLD_FPOS, 2, 19,
&               D_FLD_HELP, "Alink_B")
.
.
.
;-----------------------------------------
;Help Method

subroutine help_method
    a_ident     ,a          ;Context ID. This is what you specify
                            ; with the help qualifier for .FIELD,
                            ; the help_id argument for AX_INPUT, etc.

.include 'WND:tools.def'


proc

if (^passed(a_ident)) then
  u_htmlhelp(D_HH_HELP_CONTEXT, "myhelp.chm", a_ident)
else
  u_htmlhelp(D_HH_DISPLAY_TOC, "myhelp.chm")

xreturn
endsubroutine
```

## Implementing native Toolkit help

Toolkit includes features that enable you to easily provide context-sensitive help in your application. You define your help windows in a script file and then compile them into a window library. Toolkit enables you to specify (with a .FIELD option or a Toolkit subroutine argument) which help window to display when your user requests help. When help is requested, UI Toolkit displays the help window specified for the context and removes it when the user clicks the close button on the help window.

### Step 1 - Activating context-sensitive help

As we stated earlier, **o_help** is the reserved entry that activates the UI Toolkit help routine. You need to make this menu entry available wherever you want the user to have access to help information. We suggest that the O_HELP reserved entry be included in your General menu column.

### Step 2 - Creating the help windows

Creating the text for your help windows may be easier than you think. In fact, you may already have the bulk of the work done and simply need to reformat existing text into window definitions. For example, you may be able to borrow text directly from existing documentation, using an editor to place it in a UI Toolkit script window definition. As you will see from the example below, you can create a template for your help windows that will give all of them the same "look and feel" and yet allow for variable-length text in each window. Modifications can be made to the window contents without having to worry about reformatting the data for presentation.

### A special window called h_general

Your window script should contain a text window named **h_general**. This is a generic help window to be displayed when the UI Toolkit help routine can't find the specified help window or when no help window was specified. This window should contain a generic message that is appropriate across the entire application. If Toolkit cannot find **h_general**, it displays an error message. Your application will have a more professional look if **h_general** is defined.

Below is an example of a general help window. Although it is defined to have 10 rows, the .TRIM 1 command removes the empty lines at the bottom, leaving one blank line after the text.

```
.window h_general, 10, 40
.placement 10, 20
.text WndTxt2, position(1, 1)
There is no help available for this function.
.trim 1
.end
```

Two typical help windows are shown below:

```
.window h_lname, 10, 40
.placement 10, 20
.text WndTxt3, position(1, 1)
This field contains the contact's last name. The last name is used to
identify the record and therefore cannot be left blank.
.trim 1
.end

.window h_tk, 10, 40
.placement 10, 20
.text WndTxt4, position(1, 1)
Check the Tk box if this customer is licensed for UI Toolkit.
.trim 1
.end
```

Notice that all of the help windows are defined as being 10 rows by 40 columns and are placed at location 10, 20 in the display region. Using the .TRIM command eliminates the need for you to determine how the text fits in the window. And if you ever modify the text in the window, the resizing will take place automatically when the script file is processed.

You may want to define your help windows in a separate script file, or you may choose to place them in the script file of the window with which they are associated. If you want to use UI Toolkit's help routine as it is distributed, your help windows should reside in the library specified in U_START. Otherwise, you can replace Toolkit's help routine with your own. See the EHELP_METHOD documentation in the "Environment Routines" chapter of the *UI Toolkit Reference Manual* for the details on how to do this.

### Step 3 - Associating help windows with fields

To simplify the association of help windows with the fields in your input windows, UI Toolkit provides the ability to identify the help window for each field. You can define this association once in your repository, and every instance of that field will inherit this information. Or you can specify it directly with the .FIELD **help** qualifier. The following is extracted from our previous input window script definition:

```
.field lname, a16, pos(1, 4), prompt("Last"), pos(6, 11), fpos(6, 19), -
    help("h_lname"), hyperlink_method("contact_drill")
```

If the window named **h_lname** exists in the primary window library and the user selects the **o_help** reserved menu entry while positioned on the field **lname**, the input processor calls the Toolkit help routine, which displays the appropriate help window and waits for a user response. When the user is done reading the help window, Toolkit removes it, restores the original screen, and resumes processing the input field.

Most UI Toolkit routines that prompt the user for input have a similar mechanism to activate the help routine and display a specific help window. For example, M_PROCESS, T_EDIT, T_VIEW, and L_SELECT all have an optional argument for the name of the help window to display if the user selects the help menu entry. I_INPUT uses the .FIELD **help** qualifier. You can even have a help window for each menu entry.

# Organizing Your Display with Tabbed Dialogs

With UI Toolkit, you can organize windows and lists into a set of tabbed dialogs. While tabs are primarily a Windows feature, Toolkit makes tabs available on both Windows and non-Windows platforms. Each tab element or *dialog* contains an input window or list. All tabs in a group of tabs (called a *tab set*) are placed simultaneously, in an organized stack. Your user can select, via mouse or menu entry, any tab in the current set. The selected tab is brought to the front and its element is available to process.

### Step 1 - Create a tab set

When a tab set is processed, each member tab and its associated element (a window or list) is accessible to the user. Before a tab can be created, you must define a tab set. The example below creates a tab set. The tab set ID is returned in **contact_tab**. Each tab will be 8 rows by 71 columns. No tab element can be larger than this.

```
contact_tab = %ts_tabset(DTS_CREATE, "contact_tab", 8, 71)
```

### Step 2 - Load the input windows

Below we create three copies of the same input window, one for each of the three tabs to be created. (To create a copy of an input window, your call to I_LDINP must pass a window name in the eighth position; the load routine loads a copy of the window and assigns it the name passed in position three.) Of course a tab set doesn't need to use the same window—it can consist of a mixture of completely unrelated input windows and lists.

Next we call W_BRDR to create a title for the input window. This is important because the identifying text that displays on the tab is actually the title of the associated input window. Without a title on the associated input window, your user will have no way of distinguishing one tab from another (without selecting it).

```
for cntr from 1 thru 3
  begin
    xcall i_ldinp(contact_inp(cntr), g_utlib, "contact"+%string(cntr),
  &               D_NOPLC,,, "contact")
    xcall w_brdr(contact_inp(cntr), WB_TITLE, cont_grp[1].c_lname)
  end
```

### Step 3 - Populate the tab set

With the elements in place, we are ready to create a set of tabs. Below is the same code we used to load the input windows—with one addition. To the loop we added a %TS_TABSET(DTS_WINDOW…) function call. The DTS_WINDOW argument to %TS_TABSET creates a tab and associates an input window with it. (To create a tab and associate a list, use DTS_LIST instead.) When the user selects this tab, the window in **contact_inp(cntr)** is brought to the front, and our PROCESS_INPUT routine is called to process it. Since we have three

identical input windows, we can use the same process routine. If you need to process each tab differently (or are mixing different input windows or input windows and lists in the same tab set), you may want to specify a unique process routine for each tab.

```
for cntr from 1 thru 3
  begin
    xcall i_ldinp(contact_inp(cntr), g_utlib, "contact"+%string(cntr),
  &               D_NOPLC,,, "contact")
    xcall w_brdr(contact_inp(cntr), WB_TITLE, cont_grp[1].c_lname)
  end
tabindx(cntr) = %ts_tabset(DTS_WINDOW, contact_tab,
  &                        contact_inp(cntr), "process_input")
```

### Step 4 - Processing the tab set

TS_PROCESS is the tab processing routine. It controls the placement of the current tab and calls the appropriate tab method. Like most UI Toolkit routines, TS_PROCESS is called in an event loop. When called, TS_PROCESS *processes* the current tab: it brings the associated window or list to the front and calls the tab method. When the user selects another tab, the tab method returns to TS_PROCESS, which brings the selected tab (and its associated element) to the front and calls that tab's designated tab method. If a menu entry or button was selected, TS_PROCESS exits so your processing loop can test for and handle the exit event.

The sample below calls TS_PROCESS, and tests for an exit event. The **cont_key** method data is a group that contains an array of key values that identify the record to be loaded into each tab input window. See the code segment in step 5 for an example of how this method data is used.

If, in the tab method, the user selects the "Exit" menu entry or the "Cancel" button, TS_PROCESS loads **g_entnam** with the name of the selected event (O_EXIT or O_ABANDON) and returns immediately to this processing loop. If, on the other hand, another tab was selected, **g_entnam** is loaded with the name of the tab, in the form TS_TAB*xxxx*, where *xxxx* is the tab number. TS_PROCESS moves that tab to the front and again calls the tab method, PROCESS_INPUT, this time without returning to the processing loop.

```
tab_num = 1
repeat
  begin                     ;Processing loop
    xcall ts_process(contact_tab, cont_key)
    if (g_entnam.ne."TS_TAB")
      exitloop
  end                       ;Processing loop
```

### Step 5 - Create the tab method

The tab method below is called by TS_PROCESS for each of the tabs. We can use the same tab method because each tab contains a copy of the same input window, an input window that is processed in exactly the same way. PROCESS_INPUT starts by using the DTS_ACTIVE option to determine the current or "active" tab. (DTS_ACTIVE can also be used to change the active tab.) It

uses the tab number or "index" to determine the appropriate key to the record to be processed in the current input window. Once it has this data, PROCESS_INPUT loads it and calls the input processor. If the method determines that I_INPUT exited with the input set complete, it updates the record and returns to TS_PROCESS. Because the current tab set hasn't changed, TS_PROCESS calls PROCESS_INPUT again with the same tab set active.

If the user selects another tab, I_INPUT exits with **g_setsts** and **g_select** true; **g_entnam** contains the name of the selected tab set. PROCESS_INPUT returns to TS_PROCESS, which calls PROCESS_INPUT again with the new tab active (at the front).

If I_INPUT exits because a menu entry or button was selected, again **g_setsts** and **g_select** are true, but this time TS_PROCESS doesn't recognize the entry in **g_entnam**. It exits to the processing loop in the calling program.

```
subroutine process_input
    a_inpid         ,n
    group contkey   ,a
      key_val       ,9a77
    endgroup
.include "WND:tools.def"
.include "ids.gbl"
static record
    key             ,i4     ,1
record
    contact_dat     ,a198

proc
    key = %ts_tabset(DTS_ACTIVE, contact_tab)
    read(contact_ch, contact_dat, key_val(key), KRF=1)
    xcall i_display(a_inpid, "all", contact_dat)
    xcall i_next(a_inpid, "data", "*FRST*")
    xcall i_input(a_inpid, "data", contact_dat)
    if (.not.g_setsts)
      write(contact_ch, contact_dat)
    xreturn
endsubroutine
```

### See also

The "Tab Set Routines" chapter of your *UI Toolkit Reference Manual* for more information on using tab sets.

# Using Composite Windows to Combine Windows and Lists

UI Toolkit enables you to combine multiple windows and/or lists into a single window, called a composite window. For example, you could create a composite window that includes an input window, a table, an ActiveX control, and buttons. To the user, a composite window looks and functions as a single window.

A composite window consists of a parent window (called the composite container window) and child windows or lists. A child can be any type of UI Toolkit window or list, including ActiveX windows, tab set windows, ActiveX Toolkit lists, and other composite windows. This section contains the general steps for creating and using composite windows. For details on these steps, additional composite window tasks, and a complete example, see "Creating and Processing Composite Windows" in the "Composite Window Routines" chapter of the *UI Toolkit Reference Manual*.

### Step 1 - Create the composite container window

Start by using the DC_CREATE subfunction for %C_CONTAINER to create the composite container window. The example below creates a composite container window that

- has a Toolkit-assigned name. (Passing a null string for the *name* argument causes Toolkit to assign a unique name to the composite window.)
- fills the entire body of the application window. (Passing **g_bdysiz**, a global variable defined in **tools.def**, instructs Toolkit to use the number of rows in the body of the application window, and passing %W_INFO with WIF_SCOLS instructs Toolkit to use the number of columns in the application window.)

Additionally, this example returns the ID of the new composite container window in the **id_contain** variable.

```
id_contain = %c_container(DC_CREATE, "", g_bdysiz, %w_info(WIF_SCOLS))
```

### Step 2 - Add child windows and/or lists

To add child windows and lists to the composite window, use the DC_ADD subfunction for %C_CONTAINER. In this call, you specify the ID of the container window as well as the ID of the window or list you want to add as a child. Additionally, you can set properties for the child, including placement, tabbing position, and a menu column to be loaded when the child is processed. For example, the following adds a list that starts on the first column of the first row of the composite window and loads a menu column with the ID **id_selcol**:

```
xcall c_container(DC_ADD, id_contain, DC_LIST, id_list, 1, 1,,,,id_selcol)
```

### Step 3 - Placing the composite window

To place a composite window, place its container window. Note, however, that when a container window is placed, only child windows and lists that have been placed are displayed in the window. Unplaced child windows and lists remain invisible. Additionally, when you place a child, it remains invisible until its container window is placed. For example, the following call places the composite window whose container window has the window ID **id_contain**. If a window or list was placed, the window or list will be visible in the composite window.

```
xcall u_window(D_PLACE, id_contain, 1, 1)
```

Note that there are a couple of ways to place child windows and lists. One is to pass the *row* and *col* arguments in the DC_ADD call for the child, as in the DC_ADD example above. This instructs Toolkit to automatically place the child. The other is to use the D_PLACE operation for U_WINDOW or to use L_PLACE after the window or list has been added to the container window.

### Step 4 - Processing the composite window

To process a composite window, call C_PROCESS. This routine calls the method for the active child. Toolkit includes default composite window processing methods, but you can create your own methods and specify one in the DC_ADD call for a child. The following is an example processing loop that tests for menu entries and calls C_PROCESS, which passes the variables **chan_icontacts**, **chan_contacts**, **list_contact**, and **inp_contact** to the method for the active child.

```
;
; do_process - run input loop for composite window
do_process,
    xcall c_process(id_contain, chan_icontacts, chan_contacts,
  &                 list_contact, inp_contact)
    if (g_select)
      begin
        using g_entnam select
("O_ADD "),     call add
("O_DEL "),     call delete
("O_SEL "),     call select
("O_EXIT "),    call update    ;leaves g_select == TRUE
        endusing
      end
    return
```

### Step 5 - Removing and deleting composite windows

To remove a composite window, remove the composite container window with the D_REMOVE operation for U_WINDOW. Use D_REMOVE to remove child windows as well. For a child list, use L_REMOVE. Note the following:

- Removing a composite window doesn't remove its child windows and lists; it makes them invisible.
- Removing a child with D_REMOVE or L_REMOVE removes the child from the display but doesn't disassociate it from the composite window. To remove a child from the display and disassociate it from the container, use the DC_REMOVE subfunction for %C_CONTAINER. To delete a composite window, delete its container window. To delete a child, leave the environment it's logged in, or use the D_DELETE operation for U_WINDOW. Note that deleting a composite container window removes all child windows and lists but does not delete them.

# Using Methods

A method is a program that you write and UI Toolkit calls. You generally do not call a method directly. Instead, you give Toolkit the name of the method and let it make the call. Methods are used for a number of things: by the list processor to load an item and do item entry and exit processing; by the input processor to do field display and change processing; and by U_START to do special initialization processing—just to name a few.

## Using input methods

Input field methods are very powerful tools that enable you to set up a field prior to input or to validate input before continuing to the next field. While break processing will accomplish many of the things input methods are designed to handle, an application running on Windows cannot assume that fields will be processed sequentially. Your user could be on field A and click on field C, completely bypassing field B. If field C depends on a break activated by field B, your perfectly functional UNIX application doesn't work on Windows. But an input method is specific to the field that activates it and is totally independent of any other field.

Input methods are useful in a non-Windows environment as well. While break processing is handled within the program that calls I_INPUT, your input methods are a function of the field that activates them, handled externally to the routine that called I_INPUT. In the method, you can determine the window and field that activated it and process accordingly. This means you can write a single method for a given field, to be called wherever that field is used. You can even write a single method to perform all of your file look-ups, attach the method to any field on which you need to do a look-up, and in the method itself, determine the data and file for the look-up.

```
.field cmpny, a45, prompt("Company "), pos(2, 8), fpos(2, 16), -
 change_method("lookup_lmeth")
```

Input methods are specified as .FIELD options in a script or are registered with IB_FIELD. The sample script definition above defines a change method for the company name field. When the user completes input to this field, the input processor calls the subroutine LOOKUP_LMETH, a generic function we have created to validate input data. When LOOKUP_LMETH returns to I_INPUT, input processing resumes without returning to your input program.

In the example below, the call to I_INPUT passes arguments that are passed on to the change method. **Customer_ch** tells the change method where to perform the look-up; **cmpny** is passed so the change method can disable this field if the find succeeds. The change method returns success if the find succeeds or an error if the find fails.

```
xcall i_input(customer_inp,, customer,,,,,,, customer_ch, "cmpny")
```

When an asynchronous input method is called, **g_fldnam** hasn't been loaded with the name of the current field (the field that activated the method). Instead, an argument passed to your method contains information (and pointers to information) about the current field. (See Appendix E in the *UI Toolkit Reference Manual* for a description of asynchronous methods.)

The following code demonstrates the method for receiving the arguments passed to your change method. **A_inprec** is the data area passed to I_INPUT. **WND:inpinf.def** contains a group layout that matches the data passed as an argument to your method. We use %I_GETSTRING to get the name of the set the data will be displayed in.

```
function lookup_lmeth, ^val
    a_data_entered   ,n          ;Data entered by user
    a_data_stored    ,n          ;Data in storage format
    a_pending_status ,n          ;Result of Toolkit field validation
.include 'WND:inpinf.def'        ;Structure for input information
    a_inprec         ,a          ;Input record
    a_chn1           ,n          ;Method data parameter from i_input call
    a_disfield       ,a          ;Method data parameter from i_input call

.include "WND:tools.def"

proc
    if (a_pending_status .ne. D_OK)
      freturn(a_pending_status)
    fld_name = %i_getstring(inp_wndid, inp_fldnam)
    find(a_chnl, a_inprec, a_data_entered) [$ERR_KEYNOT=notfound,
  &                                        $ERR_EOF=notfound]
    reads(a_chnl, a_inprec)
    xcall i_display(inp_wndid, %i_getstring(inp_wndid, inp_setnam),
  &                 a_inprec)
    if ^passed(a_disfld)
      xcall i_disable(inp_wndid, a_disfld)   ;Can't modify the key field
    freturn(D_OK)

notfound,
    xcall u_msgbox('Entry not found')
    freturn(D_EMITTEDERR)
endfunction
```

The contents of **WND:inpinf.def** are displayed below.

```
group inputinfo ,a
  inp_wndid     ,i4        ;Input window ID
  inp_setnum    ,i4        ;Set number
  inp_setndx    ,i4        ;Set index (field number within set)
  inp_fldnum    ,i4        ;Field number within input window
  inp_nulinp    ,i4        ;Performing null input? (TRUE/FALSE)
  inp_fldtype   ,i4        ;Field type:
                           ; D_ALPHA, D_DECIMAL, D_FIXED, D_TEXT,
                           ; D_INTEGER, D_USERTYPE
  inp_empty     ,i4        ;Is the field empty?
  inp_setnam    ,i4        ;Pointer to name of current input set
  inp_fldnam    ,i4        ;Pointer to name of current input field
```

```
  inp_usertype ,i4         ;Pointer to user-defined data type
  inp_usertext ,i4         ;Pointer to user text string
  inp_prc_atr  ,i4         ;Input processing attributes
  inp_prc_clr  ,i4         ;Input processing color
  inp_dsp_atr  ,i4         ;Field display attributes
  inp_dsp_clr  ,i4         ;Field display color
  inp_clr_atr  ,i4         ;Field clear attributes
  inp_clr_clr  ,i4         ;Field clear color
  inp_entered  ,i4         ;Data modified?
  inp_return   ,i4         ;RETURN key pressed?
  inp_navevent ,i4         ;Type of navigation occurring (see tools.def)
endgroup
```

### See also

The *UI Toolkit Reference Manual* for information on all Toolkit methods, including input methods, list methods, and environment methods.

# 7

# Accessing Data Remotely with *xf*Server


### What Is a Client/Server System? 7-2

Defines a client/server system and identifies its benefits.

### What Is xfServer? 7-5

Explains what you can expect from your *xf*Server system operation, the basic requirements for setting up your system, and the client and server components of your system.

# What Is a Client/Server System?

A client/server system is a computer network in which one or more computers act as *clients*, requesting services or access to data, and one or more computers act as *servers*, processing the clients' requests. The primary objective behind setting up a client/server system is to set up some computers on a network as user-interface specialists and some computers as shared-data specialists.

## A basic client/server model

In a basic two-tier client/server model, the bulk of the application, including the user interface and logic processing, resides on the client machine. (See figure 7-1.) The client is usually a PC, but it can also be a workstation connected to a multiuser system. The data—as well as the logic that maintains the data's integrity—is centralized on the server. When a client application needs to access remote data, it sends the request over the network to the server.

The server can wait for client requests while processing requests from connected client applications. The computer selected to be the server must be capable of pre-emptive multitasking.

Any computer hosting a client application becomes a client. A client application is simply an application that uses client capabilities (such as the runtime included with Synergy Language version 6.1 or greater or Synergy/DE *xf*ODBC).

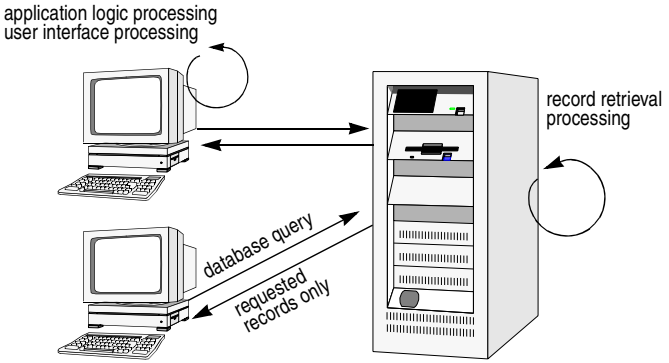



*Figure 7-1. A two-tier client/server configuration.*

## A multi-tier client/server model

The first multi-tier client/server architecture was the three-tier model (see figure 7-2), which divides processing into essentially three layers:

- User interface (collecting data from the user)
- Database serving (accessing data)
- Business logic (processing data)

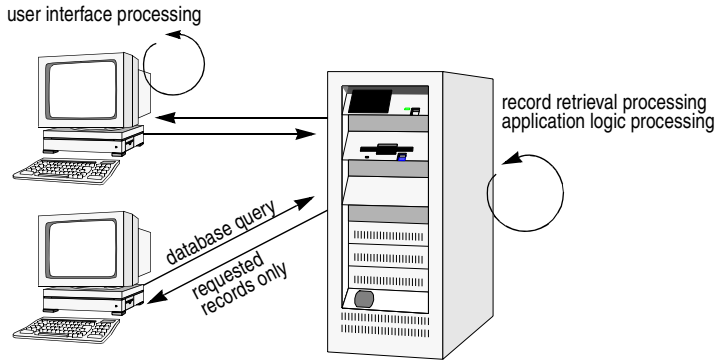



*Figure 7-2. A three-tier client/server configuration.*

The three-tier architecture led to an *n*-tier architecture, in which many application servers can be created to perform parts of the processing involved in any of the three layers, but most commonly in the application logic processing area. An *n*-tier configuration uses a distributed network and has different layers of business presentation and database access logic, each running where it is most appropriate.

This chapter addresses only the S/DE *xf*Server product and data access. For information about logic processing and the other Synergy products that enable you to create true multi-tier software solutions, see chapter 8, "Accessing Logic Remotely with xfServerPlus."

## Benefits of a client/server system

Client/server architectures are popular for several reasons:

- **Your users can share data.** This is the same benefit you get from traditional multiuser environments. Data requests and transfers between the client and the server can be concurrent and are transparent to the user. Supported file types include relative, stream, sequential, and ISAM. Any file I/O function supported by the server's "native file" system can be performed from the client application.
- **You can optimize existing systems.** When you network various types of computers in the same client/server system, you can set up each to execute the function(s) it performs best. Client/server tools enable you to use both the network connections and serial terminal connections at the same time, so you don't need a PC workstation for every user. Furthermore, a client/server system allows both the client and the server to make the most efficient use of their power:
  - The client application manages the user interface.
  - The server system handles shared file or data access requests.

  Freeing the server from interface tasks reduces processing time and memory usage.
- **You can take advantage of less expensive PCs when you expand your system's capabilities.** Today's microprocessors (PCs) rival the power of larger mainframes *and* they are faster and less expensive. Instead of adding resources when you need to add users, "offload" some or all of the system users to PCs.

# What Is *xf*Server?

S/DE *xf*Server is the technology that enables S/DE developers to add client/server capabilities to their Synergy applications. It provides the connectivity layer for remote access to corporate data from Synergy client applications. It consists of a server, which can run on Windows, UNIX, and OpenVMS, and clients, which can be Synergy applications running on Windows and UNIX.

## How the *xf*Server system operates

Let's say we have a business contact application that is used by the account managers and support representatives in a large software company. The Synergy application is located on the PCs (the client systems) of the sales and support staff, while the data about customers (because it needs to be shared) resides on the server in the MIS department. *xf*Server is what enables the server to accept client requests for data and send data back to the client PCs.

This section describes how the *xf*Server system works locally (at the client application level) and remotely (at the server level).

### How your application works locally

#### Access files using a server "address"

To access remote files, you must specify the server name in your file specifications. You can do this directly or using a logical. The direct file specification method is to append the server "name" to the filename in your Synergy Language statements and subroutines. The server name, in the form *@server,* can be either the host name of the server or the server's fully qualified domain name. An alternate form of a server file specification is *@address*, where *address* is the server's IP address on the network. For example, if your server's host name is "yubby" and the IP address is 234.52.47.128, you can specify the file using either of the following forms:

*file*`@yubby`
or *file*`@234.52.47.128`

If you are accessing a file on a separate domain, you can specify the file in the following form:

*file*`@yubby.subdomain.com`

A remote file specification cannot reference another remote file specification when the first file specification is on Windows.

We recommend that you use a logical to specify the server address rather than using a direct specification. It is much easier to maintain your code and keep up with potential changes in the client/server system architecture if your application uses logicals. Ultimately, your system will translate your file specification to the direct form and use it to access the data, but under most

circumstances, your program should not hard-code the file location. See "Getting Started with xfServer" in the "Configuring xfServer" chapter of the *Installation Configuration Guide* for more information about setting up logicals.

### I/O limitations

You are restricted to the I/O limitations of the *xf*Server system. See the *Professional Series Portability Guide* for more information on system-specific limitations.

We do not recommend using GET and PUT statements in a heterogeneous client/server environment. Remote fixed I/O (GET and PUT) is *not* transparent between client/server systems with different native line terminators (for example, Windows and UNIX). To allow for this, you may have to change client applications to support multiple operating system file types. Your client application must know where the file is located and account for either zero, one, or two line terminators when specifying the destination record size.

## How the server works remotely

### WIN, UNIX

The design for the Windows and UNIX *xf*Server platforms are similar, with only a few differences. On UNIX systems, *xf*Server operates as follows:

1. The server receives a request from a client application.
2. The server's dispatcher starts a new session server process (server thread on Windows).
3. The new session server completes the connection to the client application.
4. The session server processes the client's requests. After the process has completed all of the client's requests, the session server process (server thread on Windows) terminates. See figure 7-3.

Because Windows and UNIX servers can handle multiple clients' requests quickly, and starting a new session server process is transparent to the user, "standby" session servers are not needed. Instead, a Windows or UNIX server starts a new session server only when it receives a new client request.

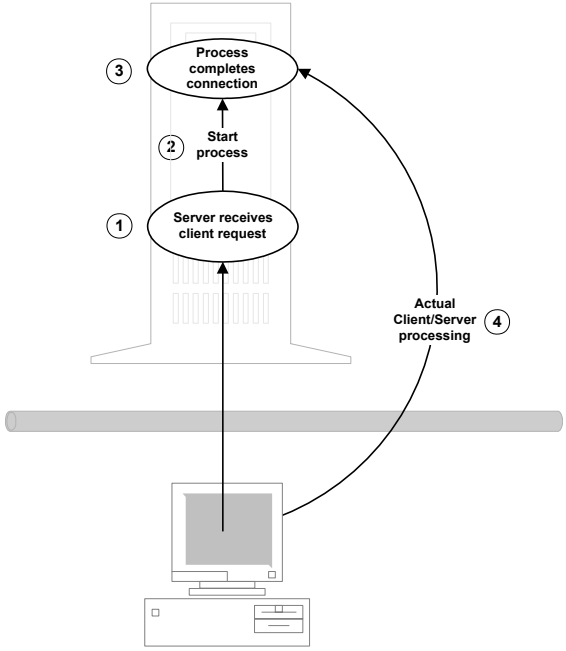



*Figure 7-3. Windows and UNIX server processing.*

### VMS

*xf*Server starts a pool of session server processes at server startup. These processes maintain and communicate their status ("I'm busy" or "I'm waiting for a client request") to the server connection manager.

As the session server pool is depleted by client requests, connection manager starts additional servers in order to maintain the free pool. If there are a sufficient number of session servers in the free pool, these temporary processes are deleted after servicing the client's request.

*xf*Server operates on OpenVMS as follows:

1. The connection manager receives a request from a client application.
2. The connection manager assigns a session server process (from the pool of processes).
3. The assigned process completes the connection to the client application.
4. The process services the client's request. During processing, the connection manager won't assign the process to another request. After the process has completed the client's request, the process sends a message back to the connection manager and waits to be reassigned. See figure 7-4.

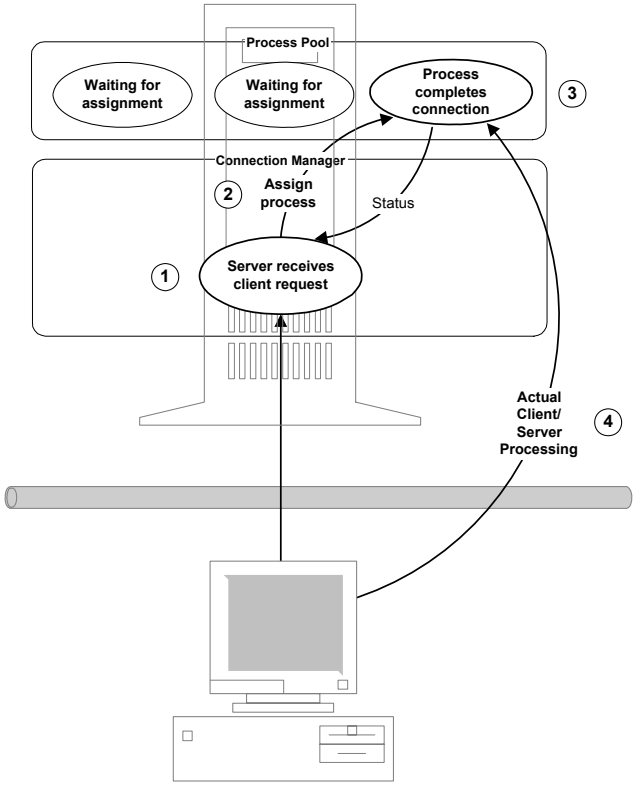



*Figure 7-4. OpenVMS server processing.*

## How the user sees the system

You may be asking yourself, “Will my users have to perform any extra steps to access the files on the server?” Your users will see the application operate exactly as it did before. Data requests and transfers between the client and the server will be completely transparent to the user.

## The TCP/IP Protocol

*xf*Server uses a fixed-port TCP/IP implementation. *xf*Server clients and servers include an option to override the port number used to connect to *xf*Server. This override enables users to resolve port conflicts and run multiple servers on a single host.

## Basic requirements

To set up your *xf*Server system, you'll need the following:

- Network hardware (network cards, cables, and connectors)
- Network software (TCP/IP) for all client and server hosts
- Your Synergy application
- Synergy Language with client support (version 6.1 or higher) for your client machine
- *xf*Server for your server system
- Log-ins and home directories for each client on the server system

### Basic *xf*Server model

The basic *xf*Server system consists of a Windows and/or UNIX client connected by a TCP/IP-based network to a Windows, UNIX, or OpenVMS server system.

## About the client and server components

### Synergy Language Client

The client portion of *xf*Server is incorporated into the runtime of Synergy Language (version 6.1 and higher) for Windows and UNIX systems. Refer to the **REL_SRV.TXT** file (included with your Professional Series distribution) for any version requirements.

### *xf*Server

*xf*Server is a background process that resides on any Windows, UNIX, or OpenVMS computer you select to be a server. Once started in the background, the server program continually accepts requests from any client computers.

The server program's executable file is **rsynd.exe**, **rsynd** (UNIX), or **rsynd.exe** and **rsdmse.exe** (OpenVMS).

**WIN**

You can optionally log information related to *xf*Server. To view the log, choose Event Viewer from the Administrative Tools window. The log of events appears under the **rsynd** source.

The event log fills up quickly. Remember to clear your event logs.

You can also monitor which files are open, who opened them, and whether they are locked by using the Monitor utility. See "The Monitor Utility for Windows" in the "General Utilities" chapter of *Synergy Language Tools*.

### UNIX

*xf*Server includes utilities you can use to manage your client/server system. To use these utilities, add the appropriate option specification when you invoke the **rsynd** command. For example, to start the Monitor function so you can track client/server activity, use the **-m** option when you execute the server command:

```
rsynd -m
```

The **rsynd** options **-m** and **-v** do *not* restart the existing **rsynd** daemon. They merely return the server version (**-v**) or enable monitoring (**-m**).

See "The Monitor Utility for UNIX" in the "General Utilities" chapter of *Synergy Language Tools* for additional information about monitoring the client/server system.

### VMS

*xf*Server includes a utility called **servstat** that you can use to manage your OpenVMS server system. See "The servstat Program" in the "General Utilities" chapter of *Synergy Language Tools* for more information.

## See also

- The "Configuring xfServer" chapter of your *Installation Configuration Guide* for additional information about setting up your *xf*Server system.
- Chapter 8, "Accessing Logic Remotely with xfServerPlus," for information about logic access.

# 8

# Accessing Logic Remotely with *xf*ServerPlus


### Overview 8-2

### What Are xfServerPlus and xfNetLink? 8-3

Describes *xf*ServerPlus and the four *xf*NetLink editions.

### Design Considerations 8-8

Discusses some of the issues you will need to consider to create a successful distributed application.

# Overview

Synergy/DE enables you to create robust two-tier and multi-tier distributed applications—including web-enabled, thin-client solutions—using *xf*ServerPlus, *xf*NetLink, and Professional Series Workbench. Using these tools, you can update your existing Synergy applications by adding a modern interface, without completely rewriting your Synergy code.

The user interface can be handled through applications written in Synergy, Visual Basic®, Java, or a .NET capable language such as C# or VB.NET. Or, you may choose to develop a web solution using Active Server Pages (ASP), ASP.NET, JavaServer Pages™ (JSP), or Web services. The *xf*NetLink edition that your application requires depends on what type of user interface you want to provide.

The application logic can be written in Synergy Language, and the database can be accessed by routines written in Synergy Language. We recommend that you use Professional Series Workbench to manage your server pages, edit your Synergy and client application code, and access Synergy type libraries, JAR files, and assemblies. (See chapter 2, "Developing Your Application in Workbench," for details about Workbench.)

See chapter 7, "Accessing Data Remotely with xfServer," for an overview of basic distributed computing and a list of its benefits.

# What Are *xf*ServerPlus and *xf*NetLink?

*xf*ServerPlus and *xf*NetLink enable you to access Synergy routines and data remotely from a Synergy, Java, COM, or .NET client. The four *xf*NetLink editions serve as clients to *xf*ServerPlus and provide the ability to access Synergy logic from a variety of front-ends. Together, *xf*NetLink and *xf*ServerPlus handle creating a connection and translating data between the client machine and your Synergy server machine.

## *xf*ServerPlus

*xf*ServerPlus is the Synergy business logic server that handles the remote execution of Synergy routines. The routines are made available for remote calling by including them in an ELB or shared image and using the Method Definition Utility to define them in the Synergy Method Catalog (SMC). *xf*ServerPlus receives a request to execute a Synergy routine from the remote client, translates the request, executes it, and returns the results to the client. *xf*ServerPlus uses the **dbs** service runtime.

## *xf*NetLink Synergy Edition

*xf*NetLink Synergy is a set of routines distributed with Synergy/DE Professional Series. These routines work in conjunction with *xf*ServerPlus to execute Synergy routines stored on a remote machine. Using *xf*NetLink Synergy, you can create a two-tier distributed application consisting of a Synergy user interface on the client accessing Synergy logic residing on a remote Synergy server. (See figure 8-1.) A Synergy front-end provides your users with a familiar look. Because the client API is provided by Synergy Language, you can use it on all supported Synergy platforms. The Synergy runtime must be deployed on the client.

There are two ways to use *xf*NetLink Synergy: %RXSUBR (and the *xf*NetLink Synergy API) and the routine call block API. You can use other Synergy/DE tools to develop your *xf*NetLink Synergy application: Composer provides drag-and-drop GUI development, as well as platform independence, and Workbench can be used to edit your Synergy code.

For more information see the "xfNetLink Synergy Edition" section of your *Developing Distributed Synergy Applications* manual. For details on making remote calls using a routine call block, see the "Synergy Routine Call Block API" chapter of the *Synergy Language Reference Manual*.

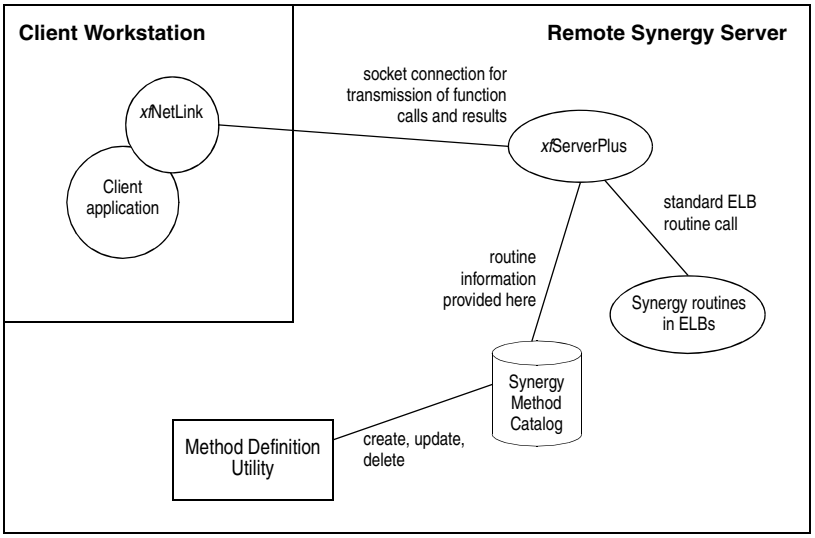



*Figure 8-1. xfNetLink and xfServerPlus in a two-tier configuration.*

## *xf*NetLink Java Edition

*xf*NetLink Java is a Java client for *xf*ServerPlus that works in conjunction with Sun Microsystems' Java programming language. Java is an interpreted language and is supported on a wide variety of platforms, including Windows, UNIX, and OpenVMS. Using *xf*NetLink Java, Synergy business logic can be accessed from any Java-capable environment, such as Java applications, JavaServer Pages, or Java applets in web pages. (We discuss two of these options below.) Together, *xf*NetLink Java and *xf*ServerPlus handle the creation of a connection between the client machine and your Synergy machine, as well as the translation of data from Java to Synergy and back to Java.

Starting with version 7.5.1, *xf*NetLink Java makes it easier to call Synergy logic from Java by implementing Java class wrappers and structure support. The *xf*NetLink Java tools enable you to create a Java JAR file that references your Synergy methods in the Synergy Method Catalog. If you're developing on Windows, you can create a JAR file from within Workbench. The JAR file presents a more natural interface for Java developers, and can be used in a Java application, JSP application, or any other Java environment.

### Java application

Using a Java application, you can create a two-tiered solution that consists of a Java application running on the client and accessing Synergy logic on a remote Synergy server. (See figure 8-1.) The Java runtime must be deployed on the client.

### JSP

JavaServer Pages are a combination of HTML and programming code written in JavaScript and Java. When a user's browser requests a JSP page, the web server executes the embedded program and returns a pure HTML page to the browser. Using JSP provides you with a three-tier distributed system, which includes an end-user machine running a web browser, a web server where *xf*NetLink and your JSP pages are located, and a Synergy server. (See figure 8-2.) JSP works with many Windows and UNIX web servers (including Apache). A servlet container for the web server is required.

Refer to the "xfNetLink Java Edition" section of your *Developing Distributed Synergy Applications* manual for more information.

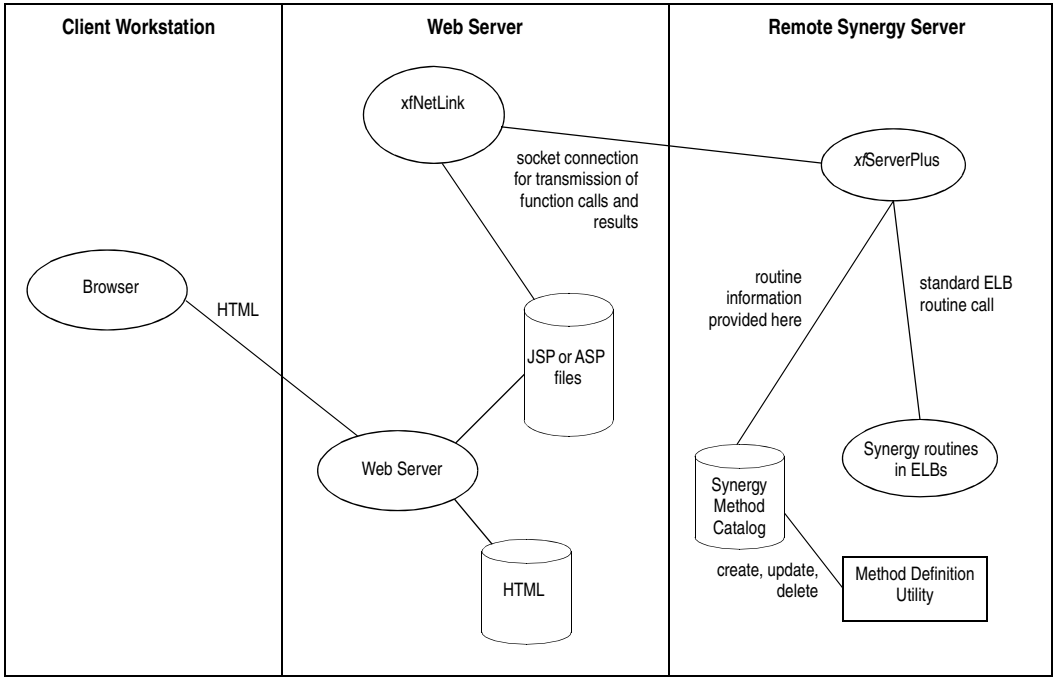



*Figure 8-2. xfNetLink and xfServerPlus in a three-tier configuration.*

## *xf*NetLink COM Edition

*xf*NetLink COM enables a client to call Synergy routines residing on a remote server machine using a standard COM calling mechanism. The user interface can be written in any language that supports COM, such as Visual Basic or Active Server Pages (ASP). Because COM was developed by Microsoft, *xf*NetLink COM is a Windows solution. (Note, however, that your Synergy routines can reside on any supported Synergy platform.)

Starting with version 7.3, *xf*NetLink COM makes it easier to call Synergy logic from ASP by implementing COM component creation and structure support. Using *xf*NetLink COM's component generation tools, you can create a COM type library that references Synergy methods defined in the Synergy Method Catalog. (These tools can be accessed from within Workbench or from the command line.) The type library you create can be used with Visual Basic, Active Server Pages, and other COM-compliant client applications to call routines residing on the Synergy server. We discuss two of these options below.

### Visual Basic

Visual Basic (VB) is a Windows front-end development tool that enables you to write applications in the Visual Basic language. Using Visual Basic enables you to develop a two-tier application: a VB application on the client accessing Synergy logic residing on a remote Synergy server. (See figure 8-1.) VB is an interpreted language, so the VB runtime must be deployed on the client.

### ASP

ASP is similar to JSP in that it is a combination of HTML coding and scripting language—either Visual Basic Script or JScript®. The page returned to the user's browser is pure HTML. Using Active Server Pages enables you to create a three-tier distributed system, which includes an end-user machine running a web browser, a web server, and a Synergy server. (See figure 8-2.) Because ASP is a Microsoft solution, it requires a web server that supports ASP, such as Microsoft IIS.

See the "xfNetLink COM Edition" section of your *Developing Distributed Synergy Applications* manual for detailed instructions on using *xf*NetLink COM.

## *xf*NetLink .NET Edition

*xf*NetLink .NET, in conjunction with Microsoft's .NET Framework SDK, enables you to create a .NET client for *xf*ServerPlus. .NET is a multi-faceted technology that provides numerous solutions. See the Microsoft website or one of the many third-party resources available for detailed information about .NET.

Using the component generation tools in *xf*NetLink .NET, you can create a Synergy .NET assembly (component) that references Synergy methods defined in the Synergy Method Catalog. (These tools can be accessed from within Workbench or from the command line.) The assembly can be used in any .NET environment to call routines residing on the Synergy server. For example, the client

might be a Windows application written in a .NET-capable language such as C# or Visual Basic .NET, or a web application that uses ASP.NET, or a Web service. We discuss a few of these options below. The .NET redistributable (i.e., runtime) must always be deployed on the Windows client machine. Although .NET is a Microsoft solution, your Synergy routines can reside on any supported Synergy platform.

### Visual Basic .NET

Visual Basic .NET (VB.NET) is a Windows front-end development tool that enables you to write applications in the Visual Basic .NET language. While VB.NET is similar to Visual Basic, Microsoft has made significant changes. (See the Microsoft web site for details.) Using VB.NET provides you with a two-tier solution: a VB.NET desktop application on a Windows client accessing Synergy logic residing on a remote Synergy server. (See figure 8-1.)

### ASP.NET

Active Server Pages .NET are similar to Active Server Pages. They are a combination of HTML coding and scripting language. For ASP.NET that language can be any Common Language Runtime-compliant language; the most commonly used languages are C# and VB.NET. Using ASP.NET enables you to create a three-tier distributed system, which includes an end-user machine running a web browser, a web server, and a Synergy server. (See figure 8-2.) ASP.NET requires a Windows web server running Microsoft IIS.

### Web services

Web services are business components that provide functionality to a web or desktop application. They serve as “black boxes”: callers use the API that you publish, and don’t need to know anything about the inner workings of the component. Web services provide you with discreet, reusable components. They are easily upgraded and can be called by multiple applications. Because Web services are a .NET solution, you can write them in any language that supports .NET; however, they can be deployed only on Windows servers running Microsoft IIS.

A Web service is not a complete solution. It is simply a component that is called by another application. Web services must be published on the web so that anyone can use them, which requires that you establish and adhere to strict security protocols.

See the “xfNetLink .NET Edition” section of your *Developing Distributed Synergy Applications* manual for detailed instructions on using *xf*NetLink .NET.

# Design Considerations

Regardless of which edition of *xf*NetLink you are using, the only way to take advantage of a distributed processing environment is to use modular, encapsulated code that can be distributed independently in a software system. If you are designing an application from the ground up, it makes sense to think in terms of completely modularizing your entire system right from the start. However, in many cases, the goal is to improve an existing application or to make several routines accessible from the Web. This makes things more complex, and the task may seem overwhelming.

The first step is to think about where you most need the benefits of modularization. What pieces of functionality do you want to make available on an intranet or the Web? What routines can you centralize into a utilities library that can be used by many or all of your applications? Do you want to have multiple user interfaces—perhaps a cell-based user interface and a web or GUI interface?

In a distributed application, processing is divided into three layers:

- User interface processing occurs on the end-user's machine.
- Business logic processing occurs on the server system. When you're building a system with *xf*ServerPlus and *xf*NetLink, this is the *xf*ServerPlus machine.
- Database processing occurs on the database server—the machine (or machines) where the data resides.

Some routines control what the user sees (for example, forms, error messages, and prompts) and other routines handle data access, calculations, validations, report generation, and so forth. When you want to display the result of a piece of application logic in the user interface, the user interface logic can call a function that performs the required task and then display the result.

For example, if you wanted to perform a log-in operation in a distributed processing environment, you would probably separate the processing layers as follows:

User interface processing:

- Draw a log-in screen.
- Perform field input.
- Validate field contents and write error messages to the screen.

Business logic processing:

- Validate the username.
- Validate that the password is correct for the specified username.

Database processing:

- Read user information from the database.

You will likely need to modify your Synergy server code before it can work effectively in a distributed environment. Once you identify the areas where modularization is needed, you can start redesigning the required functionality with your objectives in mind.

- If supporting multiple user interfaces is your priority, the first step is separating the user interface from the application logic.
- If you want to support multiple databases, the first step is separating the file access from the other application logic.
- If you plan to expose your system to your customers on the Internet, you will first need to modularize the system's application logic so that it can be called remotely from the user's browser. Another option is to write a few modular routines that perform the functions you want customers to have access to, and then integrate them into your system at a later date.

As with any other significant code modification, you can handle the move to modularity in chunks, slowly moving your system to a more and more modular state. Keep in mind that you don't need to modularize *all* of your code—just what needs to be called remotely. For specific suggestions, refer to the *Modularizing Your Synergy Code: The First Step to Distributed Computing* white paper, available on the Online Manuals CD.

## Separating the user interface from application logic

UI Toolkit, which supports event-style programming, effectively separates much of the user interface from the data management. However, Toolkit event loops are candidates for revision if you are planning to support multiple user interfaces. When you process user events using arrive and leave methods or break processing, be sure to handle those events with a call to a function in an ELB. This function must return a result that your user interface logic processes.

Error messages are an often overlooked part of the user interface. Your routines should return a success or failure status, and the user interface code should handle the actual generation of the error message on the screen. See "Handling errors" on page 8-10 for additional ideas on error handling.

## Separating data access from application logic

Separating the database from the application logic involves writing routines to handle all file I/O. This is probably the most commonly modularized area in existing applications because it hides the details of file organization, data access, and so on from the user of the routine. Since file structures change frequently, the benefits of writing routines to handle file I/O have been obvious for a long time. Again, be sure to check carefully for error messages, data validations, or other functionality that belongs in other layers of the distributed client/server system or web application model.

## Using ELBs or shared images

Executable libraries (ELBs) and shared images (on OpenVMS) provide a means of storing related routines in a common library that can be called by multiple Synergy programs. When you build a distributed system with *xf*ServerPlus and *xf*NetLink, your Synergy routines *must* be contained in an ELB or shared image and placed on the *xf*ServerPlus machine.

The ideal approach to modularizing your system is to create a set of routines with defined interfaces (formal argument lists) and then group related routines into ELBs. For example, you might group your utility routines (such as string handling and date calculations) into one ELB, your order entry application logic into another ELB, and your order entry data access into yet another ELB. These routines can be called by any application that can make use of them, particularly if you do not maintain global or common data inside the ELBs. With remote execution of ELBs, you can position each ELB where the application that uses it is located.

See the "Preparing Your Synergy Server Code" chapter of the *Developing Distributed Synergy Applications* manual for more information on using ELBs and shared images with *xf*ServerPlus.

## Handling errors

Because a distributed computing system has multiple points of failure, it is more complex to debug. Therefore, you should design your application with failure in mind.

For robust server-side code,

- trap for all possible errors on the server side.
- test your logic locally before calling it remotely. After you have modularized your Synergy code, you can test it by using the test skeleton generator utility to generate test code from your SMC definitions. It is much easier to debug your code and find problems at this point than it will be when you are calling it remotely. For more information about the test skeleton generator utility, see "Testing Your Synergy Code" in the "Preparing Your Synergy Server Code" chapter of the *Developing Distributed Synergy Applications* manual.
- return status information, in the form of status codes, as well as results, rather than displaying an error message. If you are modifying existing routines, you may want to convert them to ^VAL functions to do this, which enables you to return a status value without altering your argument list. See "^VAL functions" in the "Understanding Routines" chapter of the *Synergy Language Reference Manual*.

For robust client-side code, write your client-side code to check for errors (both *xf*ServerPlus errors and errors from your client code).

## Guidelines to improving performance and resilience

- Minimize the number of trips to the server.
- Minimize expensive operations such as starting your *xf*ServerPlus connection.
- Plan for multiple points of failure.
- When determining what to include in your client-side script, keep in mind that client-side script can be edited in a web browser or even disabled by the user. Avoid including business logic that you don't want the user to tweak.
- Favor client-side processing over server-side processing for simple field-level validations that don't require server-side data.
- Take into account that information must travel back and forth across the wire, especially for web applications. For example, you should limit the number of records displayed on the user's browser, or display them in reasonably-sized chunks, rather than sending hundreds of records.

# Glossary

**access key** — A true key in a database file, which is used to specify relationships between files.

**Active Server Pages (ASP)** — A web page that combines HTML and programming code written in Visual Basic Script or JScript. When a browser requests an Active Server Page, the web server executes the embedded program, allowing the web page to interact with databases and other programs.

**ActiveX** — A set of technologies developed by Microsoft that provides the ability to develop active, executable objects for Windows applications and web sites.

**alias** — A name that represents a directory path or a section of code. A directory alias expands to a complete directory path, which means you don't have to type the entire path when prompted for a filename or a directory. An extension-specific alias expands to the syntax for a routine or to an entire method or processing loop.

**Application window** — An area in Composer in which you can design your user interface.

**arrive method** — A subroutine called to perform special processing before an input field is processed.

**attribute** — A characteristic of an object. For example, width is an attribute of a window.

**bounds checking** — A feature of the Synergy debugger that identifies situations in which data is being stored into structures that are referenced beyond the "bounds" of their normal definition.

**break field** — A sort field on which a report break and possibly a page break is set in ReportWriter.

**check box** — An input field displayed as a small box with accompanying text to its right. When a check box is selected, an "x" is displayed in the box.

**client** — A computer hosting a client application, which is an application that requests services or access to data from another computer.

**client/server system** — A computer network in which one or more computers act as "clients," requesting services or access to data, and one or more computers act as "servers," processing the clients' requests.

**COM** — Component Object Model. The binary, interoperability standard developed by Microsoft that allows component objects developed in different languages to call one another. COM objects can be accessed by any COM-compliant application.

**command button** — The rectangular button in a dialog box that carries out a command or initiates an action (for example, OK, Help, or Cancel).

**compile-time definition** — An expression that can be completely evaluated at compile time.

**compiler directive** — A statement that instructs the Synergy compiler and is evaluated only when a program is compiled.

**compiling** — The process of translating source files containing Synergy Language statements into object files containing system-level information.

**component** — One or more related methods grouped together as a named entity.

**context** — The next input field to process OR whatever object your mouse pointer is over when you click to access the context menu in Composer.

**context-sensitive help** — "What is it?" and "What can I do with it?" information about the current context. For example, if the cursor is in an input field, context-sensitive help would provide information about what to enter in that field.

**Control Bar** — The Composer window that contains Composer's menu bar and toolbars.

**cross-reference file** — A file that contains name link associations, which ReportWriter uses to provide access to related files and data structures.

**data division** — The part of a Synergy Language program in which you define the data structures that will be used in the program.

**display method** — A subroutine called whenever the associated field is about to be displayed by Toolkit.

**distributed application** — Software whose tasks are split (i.e., distributed) between client and server machines.

**drill method** — A subroutine called when the user clicks a drilldown button (on Windows) or selects a drilldown menu entry (in non-Windows environments). A drill method is primarily used to look up additional information or display a dialog box.

**dynamic memory** — Memory that is allocated as the program requires it. The system allocates the memory and returns a pointer to the base of the memory segment.

**edit format method** — A subroutine called by Toolkit whenever the text in the field is being formatted for editing purposes.

**ELB** — See executable library.

**environment** — A program state that consists of the current definition of the screen and any terminal settings.

**environment variable** — An abbreviation defined at the operating system level for a device, directory, or path name. Also called a logical.

**escape sequence** — The set of characters emitted by a keyboard when a key is pressed.

**executable library** — A group of subroutine executable modules in a single file created by the Synergy linker.

**file stack** — A temporary file management system consisting of a single physical file that contains multiple logical scratch files accessed in a stack manner.

**font** — A design for a set of characters, comprised of typeface, point size, width, and spacing. For example, within the Helvetica typeface there are many different fonts (10-point italic, 12-point bold, and so forth).

**format** — The way data will be displayed in an input field or the way a field will be displayed in a report.

**function code** — The internal value assigned by UI Toolkit that associates a keyboard's escape sequences with menu entries.

**function name** — The name of a function code. For example, on a PC, the default name of Function 1 is "F1."

**global format** — A repository format that can be used by a field definition in any structure.

**handle** — A pointer to the base of the memory segment.

**HTML** HyperText Markup Language, the authoring language used to create documents on the World Wide Web.

**hyperlink method** A subroutine that's called when the user clicks on an input field prompt.

**information line** A single line at the bottom of the screen body that is used to display messages and general information.

**input field** A window field associated with a set of characteristics that defines how terminal input and display are to occur.

**input information** Determines how input must be entered and how it will be displayed and interpreted.

**input set** A list of one or more fields in an input window. It defines the default order for processing fields.

**input window** A window that contains one or more input fields.

**integer data** A byte-oriented, binary representation of a signed whole number.

**Java** An interpreted language used to write applications and applets.

**JavaServer Pages (JSP)** A web page that combines HTML and programming code written in JavaScript. When a browser requests a JavaServer Page, the web server executes the embedded program, allowing the web page to interact with databases and other programs.

**key** The portion of the data record that identifies the record and is used to access it.

**key map** A record that defines the correspondence between the function keys or other special keys on the user's keyboard and the functions they perform.

**key map file** A file that contains the escape sequences for each defined terminal type.

**leave method** A subroutine called to perform additional processing after an input field is processed.

**License Manager** A set of utilities that controls the installation and use of Synergy/DE products.

**linking** The process that combines one or more routines into an executable program or library.

**list** A collection of identically formatted items.

**list class** A nonvisual entity that defines the characteristics for an entire list.

**load method** A subroutine called by the list processor each time it needs a new item.

**menu entry name** The internal name for a menu entry.

**method** A program that you write and that UI Toolkit or ReportWriter calls when necessary.

**method information** Specifies any arrive, leave, drill, or hyperlink method subroutine associated with a field.

**modularization** A programming technique that requires code to be an isolated functional unit with a well-defined, published interface (i.e., an argument list).

**name link** An association between fields in different structures that is used by ReportWriter to access related files.

**object** Any window, list class, menu column, window field, input field, selection window entry, menu entry, text, line, box, or button.

**object library** A group of subroutine object modules in a single file created with the Synergy librarian. Object libraries are linked into an executable file by the Synergy linker.

**Object Manager** The Composer window that displays a visual representation of the project file, script file(s), and user interface object hierarchy.

**OLB** See object library.

**palette entry** One in a set of colors (or fonts) available to Synergy programs.

**procedure division** The part of a routine that contains the executable statements and defines the processing algorithms.

**project** A collection of one or more related files (for example, all of the files for a particular application) and the build parameters that operate on those files.

**Properties window** The Composer window that contains a list of design object attributes, followed by the property, or value, for each attribute.

**property** The value of an attribute of an object.

**quick-select character** A single character that accesses a menu column, menu entry, or selection window entry, or a key that when pressed simultaneously with the ALT key accesses a command button in a dialog box. The quick-select character is underlined on a menu or list.

**radio button** An input field displayed as a set of round buttons. The user selects one button to make a choice from among several items.

**relation** An association between structures that enables you to link the keys of one structure with the keys of the other structure.

**relational operator** An operator that compares two operands (equal to, not equal to, greater than, less than, greater than or equal to, or less than or equal to).

**rendition** A set of attributes and color of a screen display item. Rendition attributes are boldface, blinking, reverse video, and underscore.

**rendition scheme** A record that defines the display attributes and colors for your screen.

**Repository** The Synergy/DE application that orders and defines your data structures, files, and attributes.

**repository** The location where your data definitions are stored.

**reserved entry** One of a class of menu entries that is handled automatically by UI Toolkit and that is used to activate many of the standard Toolkit functions.

**runtime terminal binding** The process that binds the keyboard to the shortcuts at runtime instead of during script processing.

**script file** An editable text file that contains special script commands to define and control user interface objects and their characteristics.

**selection list** A pull-down list of options for an input field.

**server** A computer that processes client requests.

**shared image** The functionality of an executable library on OpenVMS.

**shortcut** A key or key sequence that is associated with a specific menu column entry. A shortcut provides a quick way to access a menu item, bypassing the standard menu entry selection method.

**static handle** A persistent memory handle that does not get released when the activation level of the generating routine is exited.

**string relational operator** — An operator (equal to, not equal to, greater than, less than, greater than or equal to, less than or equal to) that compares two alpha operands for the length of both.

**structure** — An entire record definition, or the combination of field and key characteristics.

**structure-specific format** — A repository format that is defined for a particular structure and can only be used by the fields in that structure.

**Synergy Control Panel** — A visual tool that enables you to customize message text, screen renditions, and key mapping.

**Synergy Method Catalog (SMC)** — Identifies the Synergy routines that you have prepared for remote calling. The SMC includes information such as the function or subroutine name, the ELB or shared image it is stored in, and the type and length of its parameters.

**tab method** — A subroutine called to process a tab in a dialog box.

**tab set** — A set of toolbar buttons that is accessed by clicking on the tab that extends above the top of the toolbar OR a group of tabs that are placed simultaneously, in an organized stack, in a dialog box.

**tabbed dialog** — A dialog box that is organized into more than one screen of fields, prompts, and buttons. Each screen is accessed by clicking the tab at the top of the screen.

**tag file** — A database that stores the information displayed for a routine when the user requests function help.

**template** — In Workbench, a method of expanding syntax automatically by generating predefined code into the editor. In Repository, a set of field characteristics that can be assigned to one or more field definitions or templates.

**text window** — A special window that contains variable-length, editable text.

**token** — A keyword that represents and is replaced with a specific name.

**type library** — A file that describes interfaces (methods and properties) exposed by objects, which allows developers to use these interfaces. Type libraries are part of Microsoft's implementation of COM.

**user-overloadable subroutine** A subroutine that you write and register so that UI Toolkit calls it automatically at the appropriate time. If you do not register your routine, Toolkit will use its default implementation.

**validation information** Defines what input is considered valid.

**volatile handle** A dynamic memory handle that goes away along with the routine activation level.

**window library** An ISAM file with a specific format to which Synergy/DE windows are saved.

**workspace** A group of related projects in Workbench.

# Index


**A**

access key 3-17
accessing memory 5-15
activating menu 6-11
Active Server Pages 8-6
Active Server Pages .NET 8-7
ActiveX Diagnostic Utility, starting 2-8
adding record 6-21 to 6-22
address, server 7-5
advantages of Synergy/DE 1-2
alias 2-22 to 2-26
  customizing 2-24
  expanding 2-17, 2-22
  parameters 2-22
Alias Editor dialog box 2-24
alias.slk file 2-22
aligned integer 5-23
allocating memory 5-13
allowable entry, specifying 3-12
alpha date, converting
  from numeric value 5-21, 5-22
  to numeric date 5-21, 5-22
Application window, Composer 4-4
application, developing 1-7, 1-9
array, defining for text window contents 6-28
arrive method 3-13, 4-15
  in distributed application 8-9
  receiving arguments 6-46
  using 6-45 to 6-47
ASP. *See* Active Server Pages
attribute 4-2, 4-5

**B**

behavior of field, defining 3-9 to 3-13
bitmap graphic on command button 4-12
blank, allowing in input field 3-12
body of screen 6-5
bounds checking 5-10
break processing 3-12, 8-9
building program 5-6
  customizing commands for 2-34 to 2-36
  from Workbench 2-31
button
  adding to input window 4-11
  bitmap graphic on 4-12
  defining 6-20
.BUTTON_SET script command 6-20

**C**

caption, window 6-4
case
  converting 3-11
  matching 3-12
CASE statement
  processing menu column 6-16
  USING vs. 5-23
change method 3-13, 4-15
channel, load method 6-32
check box
  adding in Composer 4-14
  specifying in Repository 3-10
client 7-2
  job of 7-4
  xfServer 7-9
client/server 7-1 to 7-10
  benefits 7-4
  defined 7-2 to 7-4
  two-tier model 7-2
client-side code 8-10, 8-11
closing
  Composer project 4-16
  file 6-5
  script file 4-17
code
  completion 2-15
  formatting 2-15
  generating 2-22
  template 2-26

collapsing
attribute in Properties window 4-5
files or objects in Object Manager 4-6
column
heading, in report 3-16
menu 6-4
creating 6-11 to 6-16
defining 6-11
.COLUMN script command 6-13
COM
calling Synergy routines from 8-6
creating component project 2-11
combo box, adding 4-14
command button
adding to input window 4-11
bitmap graphic on 4-12
defining 6-20
with bitmap graphic 4-12
command line, accessing in Workbench 2-5
comparing data 5-20
compile-time definition 5-23
compiling
checking errors 2-31
customizing commands for 2-34 to 2-36
from editor 5-5
from Workbench 2-31
program 5-5
script file 4-17
components in Synergy/DE 1-4
Composer 1-5, 4-3 to 4-17
exiting 4-18
help online 4-6
screen elements 4-3
starting 2-8, 4-3
terminology 4-2
using 4-8 to 4-17
composite window 6-42 to 6-44
configuration 2-13
connection manager 7-7
context 6-22
context-sensitive help 2-18, 2-19, 6-35, 6-36 to 6-38
activating 6-36
Control Bar, Composer 4-3
Control Panel. *See* Synergy Control Panel
converting
dates 5-21 to 5-23
window to text window 6-27
copying
field 3-6
object 4-5
creating
input field
Composer 4-13
UI Toolkit 6-16 to 6-20
input window 4-12
Composer 4-8
UI Toolkit 6-16 to 6-24
list 6-33
load method 6-31
menu column 6-11 to 6-16
object 4-8
script file 6-11
selection window 6-24 to 6-25
tab set 6-39
tabbed dialog 6-39 to 6-41
text field 4-14
text window 6-25 to 6-29
window library 6-11
cross-referencing data between files 3-18
#CURSOR# token 2-27
cursor, retaining position in text field 3-11
customizing
commands in Workbench 2-34 to 2-36
Workbench 2-33 to 2-40

**D**

data
comparing 5-20
cross-referencing between files 3-18
displaying to user 6-20
field, defining 3-3 to 3-6
referencing indirectly 5-18
sharing 7-4
structure, defining 6-28
unloading 6-29
data access, separating from logic 8-9
data division 5-3, 5-4
data type
integer 5-23
specifying 3-4

date
converting 5-21, 5-22
defaulting to current 3-11
performing mathematical operations on 5-21
%DATE routine 5-21, 5-22
day of the week, returning 5-21, 5-22
DBG_INIT environment variable 2-32
.dbl filename extension 5-5
dbl.als file 2-22
.dbo filename extension 5-5, 5-6
.dbr filename extension 5-6
dbs 8-3
deallocating
file from memory 6-5
memory 5-14
debugger, saving settings 2-31
debugging 5-9 to 5-11
bounds checking enabled 5-10
customizing commands for 2-34 to 2-36
project in Workbench 2-13, 2-31
restoring debugger settings 5-9
saving debugger settings 5-9
decimal point, omitting 3-11
.DEFINE compiler directive 6-6
defining
command buttons 6-20
display window 6-30
field 3-4, 3-9, 3-15
appearance 3-9
behavior 3-9 to 3-13
copying 3-6
file 3-19
input set 6-20
input window 6-17
key 3-16
list class 6-30
menu column 6-11
relation 3-18
selection windows 6-24
structure 3-3
text window 6-26
definition, compile-time 5-23
designing user interface 4-1 to 4-18
dialog box, tabbed 6-39 to 6-41
display
components, managing 6-10
information 3-9, 3-15
method 4-15
displaying
data to user 6-20
input in a field 3-8
distributed application 1-8
designing 8-8 to 8-11
performance guidelines 8-11
user interface 8-2
DM_ALLOC value 5-13
DM_FREE value 5-14
DM_REG value 5-14
DM_RESIZ value 5-13
documentation, using 1-9 to 1-10
drill method 3-13, 4-15
drilldown 4-12
duplicate value for key 3-17
dynamic
memory 5-12 to 5-17
selection window 6-25

## E

E_ENTER routine 6-8
E_EXIT routine 6-8, 6-15
edit format method 4-15
editable text field 4-14
editing
capabilities in your application 6-25
code 2-15 to 2-21
file 2-16
selecting editor 5-2
editor 2-15 to 2-21
ELB
comparing to SMC 2-9
creating 5-7
extension 5-6
using in distributed application 8-10
.elb filename extension 5-6
.END script command 6-13
END statement 5-4
ENDFUNCTION statement 5-4
ENDMAIN statement 5-4
ENDMETHOD statement 5-4

ENDSUBROUTINE statement 5-4
.ENTRY script command 6-13
entry, reserved. *See* reserved entry
enumerated field 3-12
environment
    entering new 6-8
    processing 6-8
environment variable
    benefits of using 5-18
    resetting in Workbench 2-7
    setting 2-5, 2-6, 5-19
    using 5-18
error
    compilation 2-31
    handling 6-5
        in distributed application 8-9, 8-10
event-driven programming 6-6
executable file 5-6
    xfServer 7-9
executable library
    creating 5-7
    filename extension 5-6
    restrictions 5-8
exiting
    Composer 4-18
    environment 6-8
expanding
    attribute in Properties window 4-5
    files or objects in Object Manager 4-6
    list 6-31
extension, adding to Workbench 2-36

**F**

field
    appearance 3-9
    behavior 3-9 to 3-13
    changing order in structure 3-6
    copying 3-6
    data type 3-4
    default action 3-11
    defining 3-4, 3-9, 3-15
        by copying 3-6
        for Composer 3-7 to 3-13
        for ReportWriter 3-15 to 3-21
        in Repository 6-19
    displaying 3-8, 3-10
    format 3-15
    input
        entry and display 3-10
        justification 3-10
        validation 3-11
    method subroutines 3-12
    moving 4-11
    positioning 3-10
    prompt 4-11, 4-13
    properties 4-13
    setting up prior to input 6-45
    size 3-4, 4-13
    *See also* input field; text field
.FIELD script command 6-45
#FIELD# token 2-27
file 5-5
    defining 3-19
    editing 2-16
    extension 2-36
    moving between in editor 2-21
    sharing 2-14
    stack 6-33
footer, UI Toolkit screen 6-5
format
    assigning 3-10, 3-16
    field input, defining 3-8
    global 3-8
    structure-specific 3-8
formatting code automatically 2-15
FTP 2-42 to 2-45
function
    %DATE 5-21, 5-22
    %JPERIOD 5-21
    %NDATE 5-21, 5-22
    %TS_TABSET 6-39
    %WKDAY 5-21, 5-22
    beginning 5-3
FUNCTION statement 5-3

**G**

g_entnam 6-15
g_fldnam 6-45
g_select 6-15, 6-22
g_setsts 6-22
g_utlib 6-10
generating code segments 2-22
GET statement, in client/server environment 7-6

global
format 3-8
screen components 6-9
variable 6-6

**H**

h_general window 6-37
handle, memory 5-17
header, UI Toolkit screen 6-4
help
associating window with field 6-38
Composer 1-9, 4-6
context-sensitive 6-36 to 6-38
creating window 6-37
displaying on information line 4-15
identifier, specifying 3-10
message, generic 6-37
online, implementing 6-35 to 6-38
Repository 3-2
Workbench 2-21
routine syntax 2-18 to 2-19
routines outside current file 2-19, 2-20
HTML Help 6-35
hyperlink
input field prompt 4-13
method 3-13, 4-15

**I**

I/O
limitations of xfServer 7-6
terminal 6-3
I_DISPLAY routine 6-20
I_INIT routine 6-22
I_INPUT routine 6-17, 6-25, 6-46
I_LDINP routine 6-20
I_NEXT routine 6-22
i4 variable 5-23
identifier, variable as 6-7
include file, extension 2-36
indenting code automatically 2-15
information line 6-5
specifying text on 3-10, 4-15
initialization setting 2-5
initializing input window 6-22
input
defining display format 3-8
entry and display 3-10
information 3-9
justifying 3-10
method subroutines 3-12
processing 6-21 to 6-24
saving 6-28
validating 3-11, 6-16, 6-45
input field
creating
Composer 4-13
UI Toolkit 6-16 to 6-20
data type 3-4
defining 3-4, 3-9, 3-15
in Repository 6-19
hyperlink prompt 4-13
moving
Composer 4-11
within and between 6-17
prompt 4-11, 4-13
properties 4-13
size 3-4, 4-13
input set, defining 6-20
input window
adding
command button 4-11
field from repository 4-9
line 4-11
creating
Composer 4-8
UI Toolkit 6-16 to 6-24
defining 6-17
initializing 6-22
loading 6-20
navigating 6-17
integer data 5-23
interface, designing 4-1 to 4-18
intrinsic function. *See* function
ISAM. *See* Synergy DBMS

**J**
JAR file 8-4
Java
  calling Synergy routines from 8-4 to 8-5
  creating a component project 2-11
  JAR file 8-4
JavaServer Pages 8-5
%JPERIOD routine 5-21
JSP. *See* JavaServer Pages
justifying
  data in report 3-16
  input in field 3-10

**K**
key
  defining 3-16
  duplicates 3-17
  null value 3-17
  relation 3-18
  segment type 3-17
keyword completion 2-17

**L**
L_CREATE routine 6-31, 6-33
L_METHOD routine 6-32
L_PROCESS routine 6-31
L_SELECT routine 6-31, 6-33
Language. *See* Synergy Language
leave method 3-13, 4-15
  in distributed application 8-9
  receiving arguments 6-46
  using 6-45 to 6-47
library
  executable 5-6, 5-7, 5-8
  object 5-5, 5-6, 5-8
  storing routines 8-10
  window 6-10, 6-11
line
  adding to input window 4-11
  terminator 7-6
linking program 5-5, 5-6, 5-7
  customizing commands for 2-34 to 2-36
  from editor 5-7
  from Workbench 2-31
.lis filename extension 5-5
list 6-29
  characteristics of 6-30
  combining with window(s) 6-42 to 6-44
  creating 6-33
  expanding 6-31
  loading
    input window 6-32
    item into 6-30
  processing 6-29, 6-33
  structure of items in 6-29
list class 6-30
listing file 5-5
load method 6-30
  changing at runtime 6-32
  channel requirements 6-32
  creating 6-31
loading
  item into list 6-30
  list input window 6-32
  menu column 6-13, 6-27
  text window 6-27
local screen components 6-9
logic, separating
  from data access 8-9
  from user interface 8-9
logical
  benefits of using 5-18
  setting 5-19
  using 5-18
longword-aligned integer variable 5-23
lowercase, converting to uppercase 3-11

**M**
^M data reference operation 5-15
M_LDCOL routine 6-13
M_PROCESS routine 6-14
main routine, beginning 5-3
MAIN statement 5-3
manual, accessing online 2-8
map file 5-6
.map filename extension 5-6
matching
  all characters 3-12
  case 3-12
maximum value, specifying 3-12
%MEM_PROC routine 5-13 to 5-14

memory
accessing 5-15
allocating 5-13
deallocating 5-14
defining structure 5-12
dynamic 5-12 to 5-17
registering 5-14
resizing 5-13
menu
activating 6-11
navigating 6-11
processing 6-14, 6-15
selecting from x
menu bar 6-4
menu column 6-4
creating 6-11 to 6-16
defining 6-11, 6-13
loading 6-13, 6-27
menu entry
defining 6-13
name prefixes to avoid 6-12
predefined by UI Toolkit 6-12
method 6-30
arrive 3-13, 4-15
change 3-13, 4-15
display 4-15
drill 3-13, 4-15
edit format 4-15
generating in Workbench 2-26
hyperlink 3-13, 4-15
information 3-9
leave 3-13, 4-15
subroutine, specifying 3-12, 4-12, 4-15
using 6-45 to 6-47
*See also* arrive method; leave method; load method; tab: method
Method Definition Utility 8-3
starting 2-8, 2-9
METHOD statement 5-3
minimum value, specifying 3-12
minus sign
Object Manager 4-6
Properties window 4-5
modifying record 6-22 to 6-24
modular code 8-8 to 8-9
using ELBs 8-10
moving input fields 4-11

**N**

naming
object 4-5
variables 6-7
native Toolkit help 6-35, 6-36 to 6-38
%NDATE routine 5-21, 5-22
negative, allowing values 3-12
.NET
calling Synergy routines from 8-6 to 8-7
Configuration utility. *See* Synergy .NET Configuration utility
Framework SDK version 2-38
Web services 8-7
network, hardware and software requirements 7-9
NFS-based mapped drives 2-41
numeric, converting
from date 5-21
to alpha date 5-21, 5-22
to date 5-21, 5-22

**O**

O_HELP menu entry 6-35
o_help menu entry 6-15, 6-36
object 4-2, 5-5
copying 4-5
creating 4-8, 4-13
renaming 4-5
size 4-13
object library 5-5
creating 5-6
filename extension 5-6
tip 5-8
Object Manager 4-5
OK button, adding to input window 4-11
.olb filename extension 5-6
OLB. *See* object library
online help
Composer's 1-9, 4-6
implementing 6-35 to 6-38
providing context-sensitive 6-36
Open Command tab 2-33
OPEN statement 6-5
opening file 6-5
operator, relational and string relational 5-20
order, fields in structure 3-6

**P**
paint character, specifying 3-10
Parameter Entry dialog box 2-22
philosophy of Synergy/DE 1-2
phone number, formatting 3-8
plus sign
  Object Manager 4-6
  Properties window 4-5
popping up window 6-20
portability 5-2
positioning input field 3-10, 4-11
primary key 3-17
PROC statement 5-4
procedure division 5-3, 5-4
processing
  input 6-21 to 6-24
  lists 6-29, 6-33
  menu 6-14, 6-15
  selection windows 6-25
  tab sets 6-40
product support, contacting x
Professional Series 1-4 to 1-5
Professional Series Workbench. *See* Workbench
programming
  event-driven 6-6
  tips 5-18 to 5-24
project
  Composer 4-2
    closing 4-16
  Workbench
    adding files to 2-12
    adding to workspace 2-14
    benefits 2-11
    compiling, building, and running 2-31 to 2-35
    configurations of 2-13
    creating 2-11 to 2-12
    customizing 2-33, 2-34
    debugging 2-31
    moving between files in 2-21
    opening 2-33
prompt
  hyperlink 3-13
  specifying 3-10, 4-11, 4-13
Properties window 4-5
property 4-2
  setting in Composer 4-13
protocol, TCP/IP 7-8
Prototype utility 2-9
PUT statement 7-6

**Q**
quick-select character 6-11, 6-13
quitting Composer 4-18

**R**
radio button, specifying in Repository 3-10
RDBMS, accessing 1-6
record
  adding new 6-21 to 6-22
  layout, defining 3-3 to 3-6
  modifying 6-22 to 6-24
referencing data indirectly 5-18
registering memory 5-14
relation, defining 3-18
  keys 3-16
relational operator 5-20
renaming object 4-5
reordering fields 3-6
report
  defining fields for 3-15
  displaying field in 3-15
  justifying data 3-16
ReportWriter 1-5
  defining fields in Repository 3-15 to 3-21
Repository 1-5, 3-1 to 3-21
  defining
    record layout 3-3 to 3-6
    ReportWriter files 3-15 to 3-21
    user interface characteristics 3-7 to 3-13
  displaying list of valid data 3-2
  getting help 3-2
  input fields in 6-16, 6-17, 6-19
  starting 2-8, 3-2
repository
  converting from script file 2-8
  field, adding to input window 4-9
requirements, xfServer 7-9
reserved entry
  defining column for 6-17
  loading 6-17
  prefixes 6-12
  text editing 6-26
  using 6-12

resizing
field 4-13
memory 5-13
object 4-13
restriction, executable library 5-8
routine
dispatching dynamically 5-17
displaying argument list 2-18
identifying type 5-3
moving between in Workbench editor 2-19
#ROUTINE# token 2-27
running
Composer 4-3
program 5-5, 5-8
customizing commands for 2-34 to 2-36
from editor 5-8
from Workbench 2-31
Repository 3-2
runtime 1-5
service 8-3

## S

SAVE debugger command 2-32
saving
script file 4-16
structure 3-6
work, Composer 4-16
screen
default UI Toolkit 6-4
restoring 6-8
script command
.BUTTON_SET 6-20
.COLUMN 6-13
.END 6-13
.ENTRY 6-13
.SELECT 6-25
.SET 6-20
script file 4-2
closing 4-17
compiling 4-17
converting to repository 2-8
creating 6-11
extension 2-36
saving 4-16
Script utility, starting 2-8
scrolling through text window 6-28
sdkvars.bat file 2-38
segmented key 3-17
.SELECT script command 6-25
selection list
field, adding 4-14
specifying in Repository 3-12
selection window
creating 6-24 to 6-25
defining 6-24
dynamic 6-25
processing 6-25
server 7-1 to 7-10
accessing remote files 7-5
defined 7-2
job of 7-4
*See also* xfServer
server-side code 8-11
error handling 8-10
service runtime 8-3
.SET script command 6-20
SFWINIPATH environment variable 2-33
shared image 5-8, 8-10
sharing data 7-4
shortcut 6-11, 6-13
size
field 3-4, 4-13
object 4-13
SMC/ELB Comparison utility 2-9
source code, creating 5-2, 5-3
SQL Connection 1-6
SQL OpenNet 1-6
stack 6-8
starting UI Toolkit 6-4, 6-5
static handle 5-17
string relational operator 5-20
structure
adding 4-9
assigning to file 3-20
changing field order 3-6
defining 3-3
data 6-28
memory 5-12
saving 3-6
#STRUCTURE# token 2-27
structure-specific format 3-8
subattribute, viewing or modifying 4-5

subroutine
beginning 5-3
dispatching dynamically 5-17
SUBROUTINE statement 5-3
support, contacting x
Symbol tab 2-21
syn_init_proj 2-7
syn_set command 2-6
syn_set_sdkvars macro 2-38
syn_set_synergy_ini command 2-7
Synergy Control Panel 2-8
Synergy DBMS 1-5
Synergy Language 1-5, 5-1 to 5-24
benefits 5-2, 5-3
defined 5-2
program structure 5-3
Synergy Method Catalog 8-3
Synergy .NET Configuration utility 2-9
Synergy Prototype utility. *See* Prototype utility
Synergy runtime 1-5
Synergy Type Library Configuration utility 2-9
Synergy/DE 1-1 to 1-10
advantages of 1-2
components 1-4
definition 1-2
philosophy 1-2
synergy.ini file 2-5, 2-33
synfunc 2-24
syninp 2-24
synlist 2-24
synmeappmove 2-23
synmeappsize 2-23
synmeappstate 2-23
synmecentury 2-23
synmechkfld 2-23
synmeclose 2-23
synmedspfld 2-23
synmeedtdsp 2-23
synmeentrst 2-23
synmefkey 2-23
synmehelp 2-23
synmeutils 2-23
synmiarrive 2-23
synmichange 2-23
synmidisplay 2-23
synmidrill 2-23
synmieditfmt 2-23
synmihyper 2-23
synmileave 2-23
synmlarrive 2-23
synmldblclk 2-23
synmlleave 2-23
synmlload 2-23
synmtab 2-23
synmwbutton 2-24
synmwclick 2-24
synmwclose 2-24
synmwevent 2-24
synmwmax 2-24
synmwmin 2-24
synmwmove 2-24
synmwrest 2-24
synmwscroll 2-24
synmwsize 2-24
synsub 2-24
syntab 2-24
syntax expansion in Workbench 2-17

## T

T_GETLIN routine 6-29
T_SETUP routine 6-27
T_VIEW routine 6-28
tab
method 6-40 to 6-41
set
changing active tab 6-40
creating 6-39
populating 6-39
processing 6-40
tabbed dialog 6-39 to 6-41
tag file 2-19 to 2-21
adding files to 2-20
creating extension-specific 2-20 to 2-21
custom 2-19
TCP/IP protocol 7-8
telephone number, formatting 3-8
template file 2-26
customizing 2-26
tokens in 2-27
terminal I/O 6-3
Terminal Services, Workbench on 2-7
terminology, Composer 4-2

text 6-27
  editing capabilities 6-25
  editor, selecting 5-2
text field
  creating 4-14
  retaining cursor position 3-11
text window
  adding data to 6-27
  converting 6-27
  creating 6-25 to 6-29
  defining 6-26
    data structure 6-28
  extracting input from 6-28
  loading 6-27
  setting up 6-27
  unloading data from 6-29
  viewing contents of 6-28
time, defaulting to current 3-11
tip, programming 5-18 to 5-24
title, window 4-9, 6-4
tklib.elb file 6-3
Toolkit. *See* UI Toolkit
tools.def file 6-6
.tpl file extension 2-26
TS_PROCESS routine 6-40
%TS_TABSET routine 6-39
Type Library Configuration utility. *See* Synergy Type Library Configuration utility

**U**

U_CLOSE routine 6-5, 6-6
%U_HTMLHELP routine 6-35
U_LDWND routine 6-27
U_OPEN routine 6-5
U_POPUP routine 6-20
U_START routine 6-4
%U_WINHELP routine 6-35
UI Toolkit 1-4, 6-1 to 6-47
  defined 6-3
  reserved menu entries 6-12
  screen 6-4
  starting 6-4, 6-5
uncompiled object or script 4-6
unloading data from text window 6-29
uppercasing input 3-11
user interface
  defining fields in Repository 3-7 to 3-13
  designing 4-1 to 4-18
  distributed application 8-2
    Active Server Pages 8-6
    Active Server Pages .NET 8-7
    Java application 8-4
    JavaServer Pages 8-5
    Synergy 8-3
    Visual Basic 8-6
    Visual Basic .NET 8-7
  separating from logic 8-9
USING statement
  CASE vs. 5-23
  processing menu column 6-16

**V**

validating input 6-16, 6-17
validation information 3-9
variable
  environment 5-18
    benefits of using 5-18
    setting 5-19
  global 6-6
  naming convention 6-7
  using for identifier 6-7
Variable Reference Utility 2-9
version control, accessing from Workbench 2-38
viewing
  text 6-26
  text window contents 6-28
Visual Basic 8-6
Visual Basic .NET 8-7
volatile handle 5-17
.vpj file 2-11
.vtg file 2-19, 2-21

**W**

Web services (.NET) 8-7
weekday, returning number of 5-21, 5-22
window 4-2
  composite 6-42 to 6-44
  converting to text window 6-27
  defining display 6-30
  input processing 6-16 to 6-24
  *See also* input window; selection window; text window

window library 6-10, 6-11
WinHelp 6-35
%WKDAY routine 5-21, 5-22
word processor, selecting 5-2
Workbench 1-4, 2-1 to 2-35
    accessing tools from 2-8
    adding command to menu 2-34 to 2-36
    customizing 2-33 to 2-40
    distributed applications and 8-2
    editor 2-15 to 2-21
    non-Windows development 2-41 to 2-45
    project 2-11 to 2-12
        compiling, building, and running 2-31 to 2-35
    resetting environment variables 2-7
    setting environment variables 2-6
    starting 2-4
workspace
    adding project to 2-14
    creating 2-10
    opening or closing 2-10
.wsc file 2-36

## X

xfNetLink 1-6, 8-3 to 8-7
    COM Edition 8-6
    Java Edition 8-4 to 8-5
    .NET Edition 8-6 to 8-7
    Synergy Edition 8-3
xfODBC 1-6
xfSeries 1-6
xfServer 1-6, 7-1 to 7-10
    executable file 7-9
    I/O limitations 7-6
    OpenVMS 7-7
    operation of 7-5 to 7-8
    requirements 7-9
    UNIX 7-6
xfServerPlus 1-6, 8-3
XSUBR routine 5-17, 6-15, 6-16